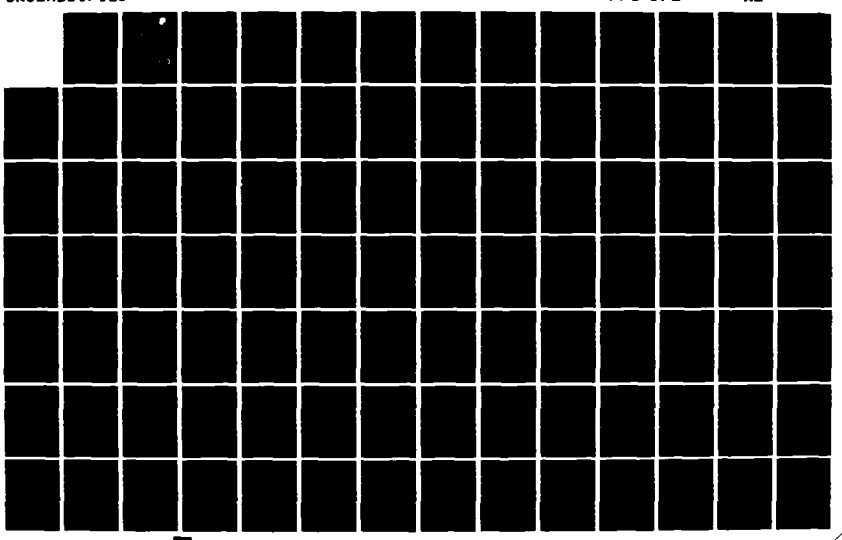
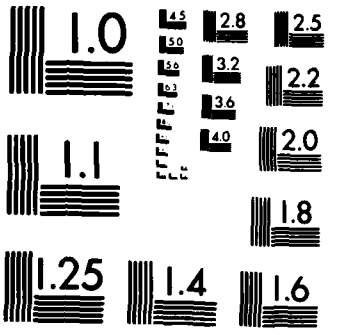
AD-A151 041 NEBULA INSTRUCTION SET ARCHITECTURE (ISA) EVALUATION 1/4
(U) DIGICOMP RESEARCH CORP ITHACA NY R D ARNOLD ET AL.
SEP 84 RADC-TR-84-190 F30602-80-C-0279

UNCLASSIFIED F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

RADC-TR-84-190
Interim Report
September 1984

NEBULA INSTRUCTION SET ARCHITECTURE (ISA) EVALUATION

Digicomp Research Corporation

R. D. Arnold, H. Boral, R. D. Cowles, A. Demers, D. J. DeWitt, J. Elkins, F. B. Schneider, M. H. Solomon and S. L. Worona

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-84-190 has been reviewed and is approved for publication.

APPROVED: Donald M Elefante

DONALD M. ELEFANTE
Project Engineer

APPROVED: Raymond P. Urtz Jr.

RAYMOND P. URTZ, JR., Technical Director
Command & Control Division

FOR THE COMMANDER: Donald A Brantingham

DONALD A. BRANTINGHAM
Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COTC) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document requires that it be returned.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) N/A	5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-84-190
6a. NAME OF PERFORMING ORGANIZATION Digicomp Research Corporation 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COTC)
6c. ADDRESS (City, State and ZIP Code) Terrace Hill Ithaca NY 14850	7b. ADDRESS (City, State and ZIP Code) Griffiss AFB NY 13441-5700
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Rome Air Development Center 8b. OFFICE SYMBOL (If applicable) (COTC)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-80-C-0279
8c. ADDRESS (City, State and ZIP Code) Griffiss AFB NY 13441-5700	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO. 64740F; PROJECT NO. 2239; TASK NO. 04; WORK UNIT NO. 01
11. TITLE (Include Security Classification) NEBULA INSTRUCTION SET ARCHITECTURE (ISA) EVALUATION	
12. PERSONAL AUTHOR(S) R. D. Arnold, H. Boral, R. D. Cowles, A. Demers, D. J. DeWitt, J. Elkins, (See Reverse)	
13a. TYPE OF REPORT Interim	13b. TIME COVERED FROM Apr 81 TO Mar 82
14. DATE OF REPORT (Yr., Mo., Day) September 1984	15. PAGE COUNT 392
16. SUPPLEMENTARY NOTATION N/A	

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB. GR.			
09	02	03	Nebula	Instruction Set Architecture	MIL-STD-1862A
09	02	04	ISA	Military Computer Family	
			MCF	Embedded Computers	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This effort conducted a review of MIL-STD-1862A (Nebula) Instruction Set Architecture (ISA). Nebula is proposed as a major ISA for embedded computer systems in the late 1980's and the 1990's. Nebula was reviewed from a number of viewpoints by independent reviewers. Part 1 of this report is a summary of the work performed and the conclusions that were reached. Included in Part 1 are an executive summary introduction, background, and a detailed summary of the conclusions reached by independent reviewers. Part 2 is a collection of the reports written by the reviewers.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Donald M. Elefante	22b. TELEPHONE NUMBER (Include Area Code) (315) 330-3241 22c. OFFICE SYMBOL RADC (COTC)

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

12. Personal Author(s) (Continued):

F. B. Schneider, M. H. Solomon, S. L. Worona

ABSTRACT

Digicomp Research conducted a review of Nebula (MIL-STD-1862A) for Rome Air Development Center. This is the final report of that effort. MIL-STD-1862A is proposed as a major Instruction Set Architecture for embedded military computer systems in the late 1980's and the 1990's. Nebula was reviewed from a number of viewpoints by independent reviewers. Part 1 of this report is a summary of the work performed and the conclusions that were reached. Included in Part 1 are an executive summary, introduction, background, and a detailed summary of the conclusions reached by the independent reviewers. Part 2 is a collection of the reports written by the reviewers.

CONTENTS

Part 2: Reports By Reviewers

PART 1

Summary of Work Performed and Conclusions Reached

Section 1

EXECUTIVE SUMMARY

AREAS STUDIED

Based on a priority list provided by RADC, reviews of Nebula were performed in the following areas and are included in this report:

* Ada1 Support
* Portability
* I/O
* JOVIAL Support
* Fault-Tolerance
* Virtualizability
* Support for Very High Level Languages, e.g. LISP, SAIL
* Data base systems support
* Multiprocessing

A section is also included which contains Nebula problem areas which arose in various discussions but which did not seem to fit into one of the above areas.

---

1 Ada is a trademark of the U. S. Dept. of Defense (Ada Joint Program Office).

GENERAL CONCLUSIONS AND RECOMMENDATIONS

MIL-STD-1862A, although being a sound 32-bit ISA in many respects, has not reached a sufficient level of maturity to easily overcome deficiencies which could prove to have a greater impact in some Air Force applications than are desirable. This conclusion is supported primarily by concerns in the following areas:

1. Through lack of clarity, through insufficient specification, and through explicit implementation dependencies, there are many problem areas relative to software portability. Although some portability issues arise due to differences between Air Force and Army procurement policies, some problems would still exist even if the Army's policies were adopted by the Air Force.
2. Given Ada's position in the DoD standardization programs, Nebula architectural support of the Ada programming language is inadequate in some respects. Features of Nebula that were designed specifically for Ada or High-Order-Languages often do not support Ada well. Compilers which attempt to use these Ada or HOL features are likely to be more complex than compilers targeted for traditional computer architectures or compilers targeted for Nebula computers which do not use the HOL features.
3. A number of the problem areas uncovered in the reviews

were a result of the development process for Nebula. It was felt that, due to the lack of development time, insufficient consideration was given to the requirements of operating systems and compilers. Rework may be required as experience is gained from writing real software for a Nebula machine.

To get from the present Nebula standard to one which is more closely tailored to Air Force requirements, starting over from the beginning may not be necessary. However, it is not sufficient to deal with the problems in isolation and apply patches as difficulties arise. Sections of Nebula should be redesigned for the attainment of specific, well defined goals. In most cases, the existing approach should be used as a base.

SPECIFIC CONCLUSIONS AND RECOMMENDATIONS

In each of the areas studied there is a list of suggested changes. These suggestions vary in emphasis from "highly desirable" to "suggested" and must be reviewed based upon the importance of the particular area in the view of the Air Force. For a summary of the recommendations, see Section 4, "Results of Independent Reviews of Nebula."

The following list consists of the suggestions (some are merely comments) which have extensive impact on the architecture:

1. The Nebula Control Board should adopt the policy of

"controlled unportability" presented in the portability report. Software visible implementation dependencies should not proliferate; where appropriate, they should be explicitly allowed. In that complete portability is impossible (or the cost/benefit ratio is too high), deviations from providing complete portability in the standard should be known and expressly allowed. An attitude of "controlled unportability" would provide documented justifications of deviations from complete portability and would allow flexibility when needed.

2. As a document, MIL-STD-1862A is not sufficiently clear, precise, or complete to be used as the definition of a ISA Standard. Regardless of the changes that finally get adopted in Nebula, the standard must be rewritten for completeness and to avoid implementation dependencies which will most certainly arise because of lack of clarity and precision.

3. Since the DoD standardization plan calls for most programs run on Nebula to be written in Ada, the Air Force would benefit if Nebula were better matched to Ada. In particular:

   * More support for Ada tasking is needed. A suggestion is included in the report.

* The procedure interface should provide more suitable support for Ada procedures, particularly, support should be included for uplevel addressing of parameters.

* More support for run-time checking would be very useful. Two minor suggestions were made, but little work was done in this area.

4. Problems with the procedure interface were discussed in several reports. More access, either structured, 2 or unstructured,3 should be allowed to the context stack.

5. The architecture is not virtualizable. While the ability to run a virtual machine monitor was viewed as highly desirable by the Computer Family Architecture evaluation.,4 it evidently was not a goal of the Nebula designers. This is rectifiable and changes are suggested to make it virtualizable. Although there is already a use for this feature,5 the Nebula Control Board should decide

---

2 By structured access we mean the addition of new instructions to give controlled access to specific information in the context stack along the lines of LPSW which loads the value of the calling procedure's PSW into a specified location.

3 By unstructured access we mean software visibility of the context stack to the normal instructions for loading, storing, and performing arithmetic and logical operations.

4 Burr, W. E., Fuller, S. H., Stone, H., Computer Family Architecture Selection Committee - Final Report, Volume II - Selection of Candidate Architectures and Initial Screening, ECOM-4527, September, 1977.

5 Statement of Work - Military Computer Family Operating System (MCFOS), Request # DAAB07-82-Q-J109, Issued by Branch D, R&D Procurement Divi-

if this functionality is indeed required and modify Nebula if that use is justified.

6. Because of its memory management system, Nebula is not particularly suited to be used as a general purpose development system.

7. Nebula will not support demand paging.6

AREAS FOR FURTHER WORK

There are several areas of Nebula which were not studied and which need to be investigated, namely:

1. The memory management system has been one of the most controversial parts of Nebula because of: its segmentation approach, its failure to guarantee to the programmer more than 16 segments in a map, and the absence of support for demand paging.
2. The IOC processor may need more processing capabilities for applications such as data base management.
3. Ways to improve Nebula support for run-time checking of constraint errors in Ada programs needs to be investigated.
4. Some features of Nebula, in particular the memory

---

sion, U. S. Army Communications - Electronics Command, Fort Monmouth, NJ.

6 This feature is also requested in the MCFOS Statement of Work.

management system, registers, and parameters should be evaluated with respect to the impact that the possible implementation techniques have on the architecture.

Section 2

INTRODUCTION

The Nebula architecture has both strengths and weaknesses. If this report seems niggardly in its praise of strong points, it is partly because they are good and do not need changing. The designers were faced with the difficult problem of using proven technology while at the same time producing an architecture which is competitive over the next 10-15 years. Nebula contains laudable features, both in creative use of existing methods and in innovative approaches to problems. However, the general emphasis of this report is on the shortcomings of Nebula, as explained below.

A LOOK TO THE FUTURE

Nebula is proposed as a major architecture for embedded systems in the late 1980's and 1990's. If adopted and accepted as a DoD-wide standard with the same committment as Ada, Nebula may rapidly become frozen and its current weaknesses could be felt through several generations of hardware and software. Therefore, Nebula must be viewed with an especially critical, though constructive attitude.

The analogy between Nebula and Ada is instructive. They are respectively the ISA and computer language for use in embedded computer systems. However, there was a competitive design process for Ada, and Nebula did not go through that process. The lengthy design and review

schedules afforded Ada were not given to Nebula. Although there were pressing historical reasons for this shortened design and review process, the effect is nonetheless observable in various weaknesses of the Nebula design.

There is now a competitive effort to build an advanced development prototype based on Nebula. However, this effort is to implement the ISA, not to evolve the ISA itself. Advancing technology and the effects of the competitive process will not be able to improve the ISA once it is frozen. This lack of improvement is especially critical in areas where the weaknesses affect software support (since architectural implementors will presumably not address such features).

Nebula is a strong and innovative architecture by today's standards. Yet, Nebula developers should not ignore the salient features of architectures evolving from today's commercial bases which are very likely to be dominant when Nebula-conforming computers are being built in full-scale production under the Army's MCF program. For example, by 1986 a 32-bit upgrade of the Motorola MC68000, with expanded memory architecture, is likely to be well established in the marketplace. Intel's iAPX 432, which already exists, is strongly oriented toward Ada. The iAPX 432 also provides significant support for multiprocessing, fault tolerance, security and protection, garbage collection and scoping of variables.7 By 1986 a revision based on experience with the iAPX 432 could be a superior alternative to Nebula, especially given Intel's evolution-

---

7 Ziegler, S., et al, "Ada for the Intel 432 Microcomputer," Computer, Vol. 14, No. 6, June, 1981, pp. 47-56.

ary approach to the market and its significant headstart. Also, militarized implementations of a range of VAX processors may exist.

The major issue is what can and should be done to develop Nebula in light of the future state of computing technology (while at the same time retaining the low-risk design characteristics of the ISA). The increased interdependence between hardware and software (especially with operating systems and compilers) is one of the most important areas currently under active research and development in industry. In the case of Nebula, this interdependence shows up in the procedure and task interfaces, including especially the context stack, parameter-passing mechanism, and exception handlng. These areas represent the least conservative of Nebula's features. They were the focus of considerable debate (and frequent revision) during the Nebula design effort, and are often cited within the Nebula Reviews as problem areas requiring further analysis.

NEBULA REVIEWS

Several versions of Nebula were used during the process of the review described below. References to "Nebula" in Part 1 refer to the draft version of MIL-STD-1862A dated "TBD" issued approximately September 30, 1981. In Part 2, the beginning of each report states which version is referenced -- most reports used the July 1, 1981 version of MIL-STD-1862A.

The reviewers were asked to answer the following questions about Nebula with respect to their area:

* What are the problems with the ISA in the area?
* What must be done to work around these problems?
* What can't be done because of the problems?
* How can the architecture be changed to solve the problem?
* What features of the ISA provide good support in the area?

ORGANIZATION OF THIS REPORT

This report is organized as follows:

1. The first part of this report contains summaries of the work performed and the conclusions and recommendations resulting from that work. The sections found in this part are:
   a. An Executive Summary which briefly describes the work performed and the conclusions reached.
   b. An Introduction to the rest of the Report.
   c. A Background section which includes chapters on: the history of the efforts surrounding Nebula; a summary of the goals of various organizations for which Nebula is expected to be a significant advance; a short technical overview of the Nebula architecture.

d. A section containing the results of the independent reviews of Nebula. This section includes chapters on: Nebula's strong features, support for programs written in Ada, writing portable programs for Nebula, problem areas and suggested changes, and areas requiring further study.

2. The second part of the report is a collection of the final reports submitted by the independent reviewers.

Section 3

BACKGROUND

HISTORY AND PERSPECTIVE OF NEBULA

Introduction

This background information on Nebula (MIL-STD-1862A) is presented to provide a perspective of the environment, and the evolution of the Nebula architecture. Topics covered include: DoD and related standardization efforts; early efforts to obtain a standard architecture; and the planned implementations of the architecture.

Standardization efforts

The DoD, the Army, and the Air Force are faced with: escalating costs for hardware and software; problems with life cycle support and maintenance; lengthy acquisition time; and overuse of old technology. To meet these problems, the military services are developing standardization plans which rely heavily on: the use of High Order Languages (HOLs) with the Ada language being used for most systems, the use of a few Instruction Set Architectures (ISAs) for most systems, and a control policy for acquisition and implementation of the architectures. Nebula is to be the major ISA for the Army and the 32-bit ISA for the Air Force. The Air Force already has a 16-bit ISA (MIL-STD-1750A) which it has been using primarily for Avionics.

Military Computer Family (MCF) Program

The development of Nebula grew out of a joint Army/Navy effort to obtain a software compatible family of military computers based on a common architecture, known as the Computer Family Architecture or CFA.[8] [9] [10] [11] [12] [13] The Naval Research Laboratory and CENTACS of the Army Electronics Command (at different times known as ECOM, CORADCOM, and CECOM) analysed basic approaches and needs. The preferred approach was to obtain a commercial architecture, if possible. Several architectures were studied, and the three finalists were: DEC PDP-11, IBM System 370, and Interdata 8/32. The DEC PDP-11 architecture was finally chosen after a lengthy evaluation process.

---

8 Burr, W. E., Coleman, A. H., Smith, W. R., "Overview of the Military Computer Family Architecture Selection," 1977 National Computer Conference Proceedings, Volume 46, AFIPS Press, Montvale, NJ, pp. 131-137.

9 Fuller, S. H., Stone, H. S., Burr, W. E., "Initial Selection and Screening of the CFA Candidate Computer Architectures," 1977 National Computer Conference Proceedings, Volume 46, AFIPS Press, Montvale, NJ, pp. 139-146.

10 Fuller, S. H., Shaman, P., Lamb, D., "Evaluation of Computer Architectures Via Test Programs," 1977 National Computer Conference Proceedings, Volume 46, AFIPS Press, Montvale, NJ, pp. 147-160.

11 Barbacci, M., Siewiorek, D., Gordon, R., Howbrigg, R., Zuckerman, S., "An Architectural Research Facility - ISP Descriptions, Simulation, Data Collection," 1977 National Computer Conference Proceedings, Volume 46, AFIPS Press, Montvale, NJ, pp. 161-173.

12 Wagner, J., Leiblein, E., Rodriguez, J., Stone, H., "Evaluation of the Software Bases of the Candidate Architectures for the Military Computer Family," 1977 National Computer Conference Proceedings, Volume 46, AFIPS Press, Montvale, NJ, pp. 175-183.

One of the criteria for the architecture was that it should be government owned. In the course of negotiation with Digital Equipment Corporation for the rights to the PDP-11 architecture, DEC announced a new 32-bit architecture called the VAX. The government tried, unsuccessfully, to secure the rights to this new architecture. Due to congressional pressure, the Navy was forced to launch development of its own program, Navy Embedded Computer System (NECS)14 and this Navy program is reportedly very similar in philosophy and approach to the MCF program.15

With the failure to obtain ownership of a commerical architecture, the MCF program had the responsibility for developing a new architecture. In 1979, Carnegie-Mellon University (CMU) began development of a new ISA which had as its major result the appearance of Nebula (MIL-STD-1862) in May, 1980.

The MCF Program now has four contracts (with IBM, GE/TRW, Raytheon, and RCA) to build Advanced Development models16 to be completed in 1983.17 After evaluation, two contractors will be chosen to produce Full

---

13 Cornyn, J. J., Smith, W. R., Coleman, A. H., Svirsky, W. R., "Life Cycle Cost Models for Comparing Computer Family Architectures," 1977 National Computer Conference Proceedings, Volume 46, AFIPS Press, Montvale, NJ, pp. 185-199.

14 Martin, Edith W., "The Military Computer Family, Part I: A Documentary," Military Electronics/Countermeasures, March, 1979, pg. 75.

15 Martin, Edith W., "The Military Computer Family, Part III: The Issues," Military Electronics/Countermeasures, May, 1979, pg. 74.

16 Statement of Work - Advanced Development of the Military Computer Family, DAAK80-80-Q-1594, U. S. Army Communications - Research and Development Command, Fort Monmouth, NJ, August, 1980.

17 At present, the family comprises three members: a 3 MIPS, 2 Mbyte super-mini version (AN/UYK-41 described by CR-CS-0034-001); a 500

Scale Development Models. In 1985, a design will be selected. A final contractor will be chosen after a competitive bidding process, and a 5-year production contract will be let. At this same time, the ISA will be reviewed and contracts for new Advanced Development models will be let which will allow for different implementations based on new technology.

Air Force Involvement with Nebula

In the last half of 1980, the Air Force joined the Army in joint control of the Nebula standard. A joint control structure was established which consisted of a Nebula Executive Board (NEB), a Nebula Control Board (NCB), and a Technical Review Committee for the NCB known as the "Tiger Team." The Air Force plans to actively pursue a standardization program using its MIL-STD-1750A ISA and JOVIAL J73 programming language while evaluating proposed applications for use with Nebula and Ada when they become available. The shift across the architectures and languages is expected to be gradual and mainly be limited to new systems or applications, or to major upgrades/modifications of existing systems.

KIPS, 1 Mbyte microcomputer version and a 500 KIPS, 128K single board version (AN/UYK-49 described by CR-CS-0035-001);

GOALS OF THE NEBULA ISA

Introduction

This section presents the role of the Nebula Instruction Set Architecture (MIL-STD-1862A) in fulfilling the goals of the Air Force System Command - High Level Systems Standardization (AFSC-HLSS) Program. A good starting point for a history of attempts to adopt a standard instruction set architecture is the Army's early efforts with the Military Computer Family (MCF) Program.

The MCF Program

Problems experienced by DoD with rapid and uncontrolled proliferation of computing systems caused the initiation of the MCF Program. Good discussions of the problems and other motivations for the MCF Program are contained in references.18 19 20 21 22 Briefly, the problems that currently exist are:

* Lack of portable applications software

---

18 Martin, Edith W., "The Military Computer Family, Part I: A Documentary," Military Electronics/Countermeasures, March, 1979.

19 Martin, Edith W., "The Military Computer Family, Part II: The Approach," Military Electronics/Countermeasures, April, 1979.

20 Martin, Edith W., "The Military Computer Family, Part III: The Issues," Military Electronics/Countermeasures, May, 1979.

21 Shohat, Murray and Edith W. Martin, "MCF Part IV: The Opportunities," Military Electronics/Countermeasures, June, 1979.

22 Brooks, et al, "MCF VI: Box vs. Module Standardization," Military Electronics/Countermeasures, August, 1979.

* Continual retraining of programmers on new machines, languages, and programming environments
* Lack of good program development environments
* Difficulty in training and retaining technicians
* Little programming in HOL's
* Old technology hardware with attendant repair problems
* Little or no competition throughout life cycle of system
* Great difficulty in upgrading systems

The goal of the MCF Program is standardization of a computer family architecture and modularization to achieve:

* Software transportability
* Meaningful competition between suppliers
* Multiple suppliers
* Graceful technology insertion
* Reduced life cycle costs

Discussions with industry have resulted in reasonable agreement in attempts to achieve these goals by standardization across four levels:

* Hardware Communication Protocols
* Higher Order Languages (e. g. Ada)
* Common hardware functionality at the box level
* Standard instruction set architectures

The above list is ranked in order, from highest to lowest acceptability. There were a number of types of standardization on which no general agreement could be reached (e.g. standardization at the module level rather than the box level). 23

Instruction Set Architecture Goals

The ISA standardization effort has in itself generated a number of goals. These goals are that MIL-STD-1862A should be:

* Implementable on a family of machines with a wide range of processing power
* An efficient host for implementing MCF HOL's (e. g. Ada)
* An efficient base for implementing MCF communications hardware protocols (e. g. 1553B, RS-232)
* A good target for implementation of a wide range of applications (e. g. real-time systems, data base systems, CCCI applications).
* Reduce the visibility of the hardware to the software.24

---

23 Martin, Edith W., "The Military Computer Family, Part III: The Issues," Military Electronics/Countermeasures, May, 1979.

24 Szewerenko, L., Dietz, W., Ward, F., "Nebula, A New Architecture and its Relationship to Computer Hardware," Computer, Vol. 14, No. 2, February, 1981.

AFSC - HLSS Program Goals

The Air Force System Command (AFSC) has a High Level System Standardization (HLSS) Program to expand the standardization of computer resources across Command programs. The goal of this program is to implement the efficient execution of Ada and very high level programming languages in a standard way by 1990. The expected benefits of this program are:

* Reduced life cycle software costs.
* Reduced logistics and training support.
* Both application and support computer programs may be used on multiple projects.
* Use of standard ISA's to facilitate hardware competition.
* Increased portability of computer programs.

The last goal, that of program portability, plays a central role in many of the other goals. In particular, life cycle costs and use of computer programs on multiple projects are tied very closely to program portability. To a certain extent, logistics and training support are also affected by portability. Part 2 of this report contains a paper which discusses portability issues surrounding the Nebula standard.

OVERVIEW OF NEBULA (MIL-STD-1862A)

Purpose

The following paragraphs present the major points of the Nebula Instruction Set Architecture (ISA). They are intended for the newcomer to Nebula and assume no prior knowledge of the ISA. The goal is not to define the ISA in detail but to give an overall picture of the architecture and to provide a background for understanding the reports in Part 2.

Basic Concepts

The Nebula ISA is a 32-bit general register architecture with byte addressable memory. Instructions are sequences of bytes with the first byte specifying the operation and the succeeding bytes specifying the operands. An operand specifier is a sequence of one or more bytes specifying the location and size of an operand. The operand specifiers provide a variety of addressing modes available in multiple forms of varying compactness. Data types such as logical, real and integer are available in several sizes: integers and logicals may be 8, 16, or 32 bits; reals may be 16, 32 or 64 bits. Operands of a given instruction may be of mixed sizes. Size conversions are performed automatically.

The architecture provides a procedure-based control structure. Procedures may be invoked by calls, interrupts, traps, supervisor handled exceptions, vectored supervisor service calls, or as independent tasks. The procedure mechanism provides for parameter passing and maintenance of control linkages. Each procedure has a procedure context containing

its own register set of up to 16 registers (one of which is subsumed by the program counter), its parameters, its PSW, and the state of its exception handler. Stack frames for procedure contexts are allocated on stacks called context stacks which are protected from any access other than through the procedure interface. Nebula instructions use virtual addresses that are translated using a hardware supported memory mapping scheme. The segmented virtual memory provides at least 16 variable size segments mapped through an associative memory segment table which contains access rights and relocation amount. One half of the virtual address space is accessible only to the supervisor.

Addressing Operands

Nebula accesses operands by operand specifiers. Each is evaluated to give the location and size of an operand. Nebula provides about 8 addressing modes depending on how one groups the operand specifiers. These modes are as follows:

* Literal Mode specifies an instruction stream constant.
* Register Mode designates a register in the current procedure context.
* Indirect Register Mode designates a register (other than register 0) which contains the address of the operand.
* Register Indexed Mode designates a register and a signed displacement. The signed displacement is added to the contents of the register to form the address of the operand.

* Absolute Mode provides the 32-bit absolute address of the operand.

* Parameter Addressing Modes allow access to parameters defined by the caller of the procedure. Access is by giving the number of the desired parameter in the parameter list. Short Parameter Mode encodes the parameter number (in the range 1 - 7) in the operand specifier. Extended Short Parameter Mode allows access to all 255 possible parameters by using an additional byte to indicate the parameter number when the Short Parameter Mode indicates access to parameter 0. Long Parameter Mode sets the number from another operand specifier embedded in its specifier. Because an operand specifier may be embedded in another, this mode is called a "compound" mode (only noncompound specifiers may be embedded in a compound mode's specifier). For example in Long Parameter Mode, the parameter number may be described as the contents of a register by a Register Mode operand specifier; this mode is necessary if the parameter number is not known at compile-time.

* Unscaled Index Mode is another compound addressing mode. The operand specifier includes two operand specifiers for the index and the base. The base and index specifiers are evaluated and combined to determine the address of the operand.

* Scaled Index Mode is similar to Unscaled Index Mode. It is also compound and includes index and base specifiers. However, the index is scaled before adding it to the base. The scaling is performed by multiplying the index by the size of the operand (1, 2, 4, or 8 bytes) as indicated in the operand specifier for the base.

A Procedure's Local Context

Unlike conventional architectures, Nebula has no common set of general purpose registers accessible to all procedures; each procedure has its own set of general purpose registers. When a procedure is invoked, a frame (procedure context) for that procedure is allocated in memory on a context stack. This frame contains the general purpose registers for that procedure, the PSW of the procedure, its program counter, information about any parameters it was passed and optionally the address of an exception handler. A procedure may have up to 16 local registers. Register 0 is the program counter for the procedure.

The integrity of the context stack is maintained by not allowing access to its contents except through the procedure interface. This interface includes the procedure-call mechanism, parameter and register addressing modes, exception handling instructions, and program counter manipulation. Access to other procedure contexts is restricted to reading the PSW of the caller, and access to any register passed as a parameter.

In addition to the information on a procedure's context stack, various fixed characteristics of the procedure are described by a procedure descriptor located at the procedure's entry point. This descriptor tells whether the procedure has a variable or fixed number of parameters, the number of parameters (if the number is fixed), and the number of local registers the procedure is to have. The Processor Status Word (PSW) also contains information about the current procedure. It includes the number of registers accessible to the procedure, the number of parameters the procedure was passed, and other state information. The context for each procedure includes the PSW which defines its capabilities.

The architecture supports two active context stacks corresponding to the Kernel Context Pointer and the Task Context Pointer. The hardware remembers which context stack is active; the current stack is changed by traps, interrupts, and tasking instructions. Each task in the system has associated with it a unique context which is in turn associated with the Task Context Pointer. An instruction is provided to change the active task context stack (LTASK).

Procedure Invocation

Procedures are invoked by executing calls. A call specifies an entry address and a parameter list. The parameter list consists of a sequence of operand specifiers. The procedure invocation establishes a correspondence between the operand specifiers in the parameter list of the caller and the parameter addressing modes of the called procedure.

Within the called procedure, a parameter is accessed by an operand specifier using a parameter addressing mode.

The operand specifiers of the caller are evaluated to yield a location and size. The corresponding parameter addressing modes are bound to this location and size information. Thus all parameters are call-by-reference.

The manner of specifying the entry address depends upon the type of call. Normal calls specify the entry address as an operand. Supervisor calls and unimplemented instructions obtain the address by indexing into protected tables of addresses. Interrupts and traps are assigned fixed vectors.

An OPEX trap is defined for instructions that are not implemented in hardware or microcode; the trap may pass control to a software procedure which then simulates the instruction. The difference between a microcode implemented instruction and an unimplemented instruction code that invokes a software procedure performing the same operation is not detectable in the object code of the machine.

Memory Management

Nebula instructions use virtual addresses. These are translated to physical addresses by the memory management system. The memory management system divides the virtual address space of the supervisor and each task into a number of segments of variable size. Each segment begins at a variable address in virtual address space described by a table in physical memory. Segments may be protected against various types of

access (see the next section on protection codes) and may be declared accessible only by processes which are executing in privileged mode. The supervisor can directly access the segments of the current task subject to these restrictions.

The virtual address space is divided into two halves. One of these halves is accessible only when the processor is in supervisor mode. There are two memory maps available for address translation, a supervisor map and a user map. An address in the supervisor half of the virtual address space uses the supervisor map for address translation. Addresses in the user half use the user map.

These maps are pointed to by the Supervisor Map Pointer Register and the User Map Pointer Register respectively. Each register contains, in addition to the map pointer, a bit indicating whether or not relocation is to be performed.

map-pointer-register ::= < address-of-map, relocate >

Each memory map is a sequence of segment descriptors. Each segment descriptor contains a virtual address bound and a signed relocation amount. The sequence is ordered so that the virtual address bound in each descriptor is strictly greater than that of the preceding segment descriptor.

memory-map ::= < segment-descr1,...,segment-descrk >

segment-descr ::= < virtual-addr-bound, reloc-amt >

An interpretive semantics for address translation is as follows.

1. Given an address, determine in what half of virtual memory it lies and select the appropriate map pointer register.
2. If the register indicates that no relocation is to be performed, the physical address is the same as the virtual address.
3. If relocation is to be performed refer to the map indicated by the map pointer register.
4. Select the appropriate segment descriptor in the map by comparing the virtual address (upper) bound of the segment descriptor with the given virtual address.
5. The physical address is then the sum of the virtual address and the relocation amount.

Protection

The description thus far has ignored protection features and checking of access rights. To understand the protection mechanism, it is necessary to elaborate on the structure of the architecture.

The processor is always in exactly one of supervisor or task mode; also, it is always in exactly one of privileged or unprivileged mode. The PSW determines what combination of these modes is in effect.

The map pointer registers contain a protection flag (in addition to the map address and relocate flag already described). The protection flag indicates whether or not access checking is to be performed. Seg-

ment descriptors contain a privilege flag and a protection key (in addition to the virtual address bound and relocation amount).

```
map-pointer-register  ::= <  address-of-map,
                             relocate,
                             protect  >

segment-descr  ::= <  virtual-addr-bound,
                      relocation-amount,
                      privilege-flag,
                      protection-key  >
```

Taking protection into account, address translation proceeds as follows.

1. Given an address, determine in what half of virtual memory it lies and select the appropriate map pointer register.
2. If it is in the supervisor half and the processor is in user mode a trap (Invalid.supv) occurs.
3. If the register indicates that no relocation is to be performed, the physical address is the same as the virtual address.
4. If either the relocation or protection bit of the map pointer register is set, select the appropriate segment descriptor in the map by comparing the virtual address (upper) bound of the segment descriptor with the given virtual address.

5. If protection is enabled, perform checking for privilege violations and access violations. A privilege violation takes place if the segment descriptor is marked privileged and the processor is not in privileged mode. An access violation takes place if the protection key of the segment descriptor does not allow the type of access being performed.

6. If no protection violations occurred, relocation is performed by summing the virtual address and the signed relocation amount.

The following access restrictions are available as protection keys:

* No Access -- any access shall cause a trap.
* Instruction Access Only -- instruction fetch and reading of literals is allowed.
* Data Read Only -- reading of operands is allowed.
* Instruction or Read Access -- instruction fetch, literal fetch, and reading of operands is allowed.
* Data Read/Write -- reading and writing of operands is allowed.
* Context Only -- access as part of a context stack is allowed. Any other access shall cause a trap.
* Reserved1 -- any access shall cause a trap.
* Reserved2 -- any access shall cause a trap.

Context stacks can only be allocated in segments with a protection code of Context Only. Context segments cannot be manipulated directly by any instruction. Proper manipulation of this information is enforced by only allowing access through the procedure interface.

Input and Output

The primary I/O interface is through a special processor called an Input/Output Controller (IOC). The IOC is a 16 bit processor with a standard instruction set tailored to the requirements of three communications interfaces: the Air Force 1553B I/O bus standard, Serial Point-to-Point (SPP), and Parallel Point-to-Point (PPP). Interaction with I/O devices through an IOC is accomplished by accessing control registers as locations in the I/O portion of physical address space (the first 2**20 locations in memory). The IOC's access memory through the virtual addressing mechanism. Each IOC program (also referred to as a channel program) has associated with it three virtual segments, one each for instructions, data, and messages. The memory management system performs some checking to see that these segments are used as intended, and that I/O program specified addresses lie within the specified segments. For a more detailed discussion of the I/O architecture in Nebula, refer to the paper in Part 2 covering that topic.

Section 4

RESULTS OF INDEPENDENT REVIEWS OF NEBULA

INTRODUCTION

This section summarizes the conclusions and recommendations of the reviews presented in Part 2. The comments in the reviews range from:

* Strongly suggested changes,
* Possible changes depending on the importance that the Air Force puts on the application area,
* Problems that are raised but for which no solution is suggested,
* Mismatches between the application and Nebula that can either be programmed around or can result in modification of the architecture,
* Comments on requirements or useful features of the operating system or other system programs for Nebula to support the various application areas,
* General comments on the architecture.

This section does not present all of the results of the reviews. For complete lists of results and for the details and rationale of the conclusions presented in this section, the reader should read the actual papers in Part 2 of this report.

Notational conventions used to refer to the papers in Part 2 as shown below. Often a page number is included in the notation (e.g. [PORTABILITY p. 15]).

[ADA] -- Implementing Ada on the Nebula Architecture: Design Issues and Alternatives

[PORTABILITY] -- Nebula and Portability

[I/O] -- Analysis of Nebula Architectural Support for I/O

[JOVIAL] -- JOVIAL/Nebula Suitability Report

[FAULT-TOLERANCE] -- Building Fault-Tolerant Systems with Nebula

[MULTIPROCESSING] -- The Nebula Architecture and Multi-Processor Systems

[VIRTUALIZABILITY] -- Nebula Architectural Support for Virtual Machines

[VHLL] -- Suitability of Nebula for Very High Level Languages

[DATABASE] -- Evaluation of the Nebula Processor for the Implementation of Database Management Systems

[MISC] -- General Observations on Nebula

GENERAL COMMENTS

General Architecture Evaluation

The general type of architecture chosen was not addressed by this review. Research is being done in many directions on architectures: High Level Language Architectures,25 Intermediate language architectures, e. g., the Western Digital Pascal Microengine,26 capabilities architectures, e. g., the Intel iAPX 432,27 reduced instruction set architectures,28 and distributed architectures. While these approaches do not in general meet the Nebula requirements of a low risk design, architectures of these types may be in widespread use within the same time frame that Nebula computers are planned to enter the full-scale production phase. This should be an incentive to produce as excellent a design as possible. The Nebula designers chose a much more conventional

---

25 DiIenno, T., and Chu, Y., Evaluation of Nebula Architecture for JOVIAL Data Contructs, Dept. of Computer Science, Univ. of Maryland, Tech. Report TR-1051, USAF grant AFOSR-79-0056.

26 WD/90 Pascal Microengine Reference Manual, Western Digital, Newport Beach, CA, 1979.

27 Introduction to the iAPX 432 Architecture, Intel Corportaion, Santa Clara, CA, 1981.

28 Patterson, D. A., Ditzel, D. R., "The Case for the Reduced Instruction Set Computer," ACM SIGARCH Computer Architecture News, Vol. 8, No. 7, 1980.

architecture; in fact, Nebula is similar in many respects to a VAX.[29] The most likely consequences of this choice are inefficient support for security and, if Ada is successful, weak support for Ada (relative to architectures of the day which can be designed for Ada using experience gained from Ada usage). In any case, the general type of architecture chosen was not evaluated.

Other Comments

There were several comments mentioned in the Introduction, to summarize:

1. The design of Nebula was not time-phased very well with the design and implementation of Ada — the language that is expected to predominate on Nebula machines. When Nebula was designed, there was no experience with Ada programs. Neither an implementation, nor even a lasting design of Ada existed.
2. To the degree that the low-risk design criteria is met, the capabilities of Nebula should be considered in light of expected capabilities of architectures in the late 1980s and 1990s rather than to architectures in use today.
3. The Nebula design had the luxury of not having to exhibit

---

29 The VAX Architecture Handbook. Digital Equipment Corp., 1981.

any type of compatibilty with any existing software. Future modifications of the architecture are likely to be severly restricted due to software compatibility considerations -- even when modifications are being attempted to correct parts of the design shown to be inadequate. There should be a high degree of confidence with the architecture before it is finalized due to the relative ease and low cost of changes made now rather than later.

4. With respect to the research in architecture, it should be kept in mind that while advances in implementation technology can perhaps be inserted into future generations of Nebula computers, advances in architectural technology, for the most part, will not be easily inserted since they are more likely to impact program portability.

Implementation Problems

This report did not consider implementation in its evaluation. Thus implementation problems, and more importantly, the effects of possible implementations on certain features of the architecture were not analyzed.30

---

30 Both the memory management system and the treatment of registers implicitly assume non-standard implementation techniques.

NEBULA'S STRONG FEATURES

Much of this report deals with criticism of Nebula. Little has been mentioned about the positive aspects of the ISA. While Nebula has assets, the problem areas are precisely what needs to be considered for impact and possible modification of the ISA. This section presents some of the more praiseworthy features of Nebula. The praises for these features, however, are qualified; the features introduce, directly or indirectly, many limitations.

Goals of Nebula's Designers

The goals of Nebula and of the designers[31] [32] are, in general, laudable. Some of these goals are discussed in Section 3; others were adherence to the following principles of architectural design:[33]

* Regularity
* Orthogonality
* Composability
* "One versus all"

---

31 Szwerenko, et. al., "Nebula: A New Architecture and its Relationship to Computer Hardware," Computer, Vol 14, No. 2, February, 1981, pp. 35-41.

32 Dietz, W., "Nebula Design and Rationale", Carnegie-Mellon University, a paper from Proceedings of Panel on Effect of Nebula ISA on Ada, edited by Martin, E., Hammond, Major D., 15th Annual EIA G-33 Committee Computer Resources, Data and Configuration Management Workshop, September, 1981.

33 For an informal discussion of these, refer to Wulf, W., "Compilers and Computer Architecture," Computer, Vol. 14, No. 7, July, 1981, pp. 41-47.

* "Provide primitives, not solutions"

However, as discussed in [PORTABILITY] and [MISC], the goals are not always attained.

Instruction Format

One of the most sound and carefully thought out areas of Nebula is the instruction format. There are two general features worth mentioning here: the size of a memory operand is encoded in the operand specifier, rather than in the opcode; binary operations are usually provided in both two and three operand formats. Encoding the size in the operand specifier rather than the opcode, and providing many binary operations in the three operand format may result in many benefits. The size encoding is probably Nebula's most noteworthy feature. It can drastically reduce the total number of opcodes needed. It also makes the compiler writer's task easier by largely eliminating the optimization analysis which determines what sizes to allocate to variables and when to do size conversions.

In actual fact, the ISA does not meet the regularity or orthogonality conditions sometimes claimed [PORTABILITY], [MISC]. The ISA is not regular with respect to these features because there are binary arithmetic operations that exist only in the three operand format and because all operands are not operand specifiers (this happens in 25% of the instructions). The size of an operand specifier is not orthogonal to the opcodes both because all operands are not determined by operand specifi-

ers, and because the opcode interprets some operand specifiers as address operands.

Nebula's Support for HOL Procedure Interfaces

Nebula's procedure interface provides support for HOL procedures and provides a mechanism which many different parts of Nebula (task initiation, interrupts, vectored operating system calls, exceptions, traps and OPEXs) can use. Many of the reports discuss problems with this interface. Because of the commitment to using exactly the same interface for many different functions, the designers were faced with making tradeoffs between the contradictory demands for quick interrupt response and for providing as much help as possible for HOL procedures calls.

Enhanced I/O Security

Another feature of Nebula is the use of virtual addresses in the I/O controllers to enhance security. It may be the most helpful, and the only really innovative, feature of Nebula for supporting security. Because the data buffer, message pointer and channel program must each lie within a segment, there are two unfortunate consequences. First, as explained in the report on multiprocessing, this doesn't allow the channel to do scatter/gather I/O message processsing into different users' processes without the intervention of the CPU. Second, as mentioned in the virtualizability report, it restricts the flexibility of the CPU in mapping segments when setting up segments for a virtual machine.

NEBULA'S SUPPORT FOR ADA

ADA SUPPORT

Efficient support for Ada should be one of the two most important concerns in evaluating the Nebula architecture. The paper in Part 2 is not a general analysis of Nebula's support for Ada, but is actually the third in a series of reviews. It takes up where the earlier reviews left off. Further evaluation of Nebula and Ada was also done and is discussed here and in [MISC]. The first review[34] [35] was written by Mark Davis of Intermetrics under contract to CECOM in February of 1981. This report pointed out areas where there were no problems (e.g. support for Ada's data types) and discussed several problem areas (e.g. uplevel references of parameters, copy versus reference parameter passing, Ada tasking). Some minor changes to the architecture were subsequently made by the Nebula Control Board,[36] but most of the problems raised were not resolved. The second review was part of the general review of Nebula conducted by the EIA from January to April of 1981. The Ada portion of this review was written by Roger Arnold of Boeing Aerospace Corpora-

---

34 Davis, M., Nebula as a Target for Ada, IR #655 Intermetrics, February, 1981.

35 Mark Davis wrote an update to this report in June ("Nebula as a Target for Ada: Summary and Update", June, 1981). This summarized the changes to Nebula concerning Ada, reviewed the findings of the original report, and discussed the author's changes in viewpoint on some of the problems.

36 These changes involved treating parameters more uniformly (previously there were restrictions on accessing parameters after number 7) and several changes to the exception handling mechanisms.

tion.[37] [38] A consensus was reached on the following points in the review:

1. The context frame/call mechanism must support "cactus stack".
2. Display based addressing is "highly desirable".
3. The uniform treatment of parameters and local data is "highly desirable", both with respect to modes of access and uplevel referencing.
4. Better range checking is needed.
5. Task management and intertask communication needs study.

The following list is the prioritized list of issues for further consideration that came from this EIA review:

1. Multitasking within a common memory space
2. Uplevel referencing of parameters
3. Uplevel referencing of variables
4. Range checking
5. Inter-task communication
6. Copy versus reference parameter passing

---

[37] Martin, E. W., Fischer, H., Fagg, A., Arnold, R. D., James, J., "Report of MIL-STD-1862 REVIEW COMMITTEE d.b.a. NEBULA REVIEW COMMITTEE", Electronic Industries Association G-33 Data and Configuation Management Committee Computer Resources Task Group, June 1981.

[38] The following paper was also written as part of this review: Arnold, R. D., "Ada and the Nebula Architecture", Boeing Aerospace Corp., February, 1981.

7. Joint context and data frames

8. Memory protection

9. Referencing parameters greater than 7

10. Necessity of the compound parameter mode

Of these two lists, only the issue on referencing parameters greater than 7 resulted in a change to Nebula.

One of the areas that received little attention in the two earlier reviews was the implementation of Ada tasks on Nebula. The paper in Part 2 of this report primarily investigates this area. Several possible implementation strategies are discussed; they are shown to be generally inadequate. Two sets of changes, one upward compatible and one incompatible with the present version of Nebula, which give Nebula better support for Ada tasking are then discussed [ADA p. 13].39 Besides supporting tasking in Ada the incompatible changes also provide some support for the data frame.

The following is a prioritized list of the suggestions of this report concerning Ada:

1. Modify CALL and RET instructions, and add LIMITED CONTEXT SWITCH instruction for Ada tasking.

2. Modify integer truncation such that if a truncation

---

39 Also, see the subsection on Ada tasking later in this section.

exception is to be raised as a result of an operation, then the result won't be written to memory. The instruction will be aborted and the exception will be raised.

3. Add an instruction analogous to RANGE with floating point operands.
4. Consider the addition of more support for range checking.
5. Add hardware support for the data stack. This may include several of the following:
   a. Hardware allocation of the data frame
   b. Hardware supported display and display based addressing modes
   c. Remove parameters from context stack and incorporate them on data stack. The parameters may still be passed by the call instruction, but the parameter addressing modes would be deleted. Parameters would be accessed in the same manner as local data.
   d. Include support for uplevel referencing of data and parameters.
6. Provide more access to information in suspended procedure contexts. If changes from the previous point don't allow uplevel parameter references, then changes made under this point should.

Points 1 and 2 are discussed in [ADA] and [MISC] respectively. Points 3 and 4 concern hardware support for run-time checking. This is an area

where languages need hardware support and aids for compiler optimization. Because of inefficient support for run-time checking in implementations, there is a tendency to turn the feature off, thus negating a very useful feature in Ada. One problem with providing support for this with Ada, is that there is a large measure of uncertainty: in how much optimization the software can do, to what extent programmers will remain within constraints,40 and as to how efficient will the hardware solution be. In any case, Nebula should consider providing more support, but not in a way which interferes with the ability of the compiler to optimize away a particular run time check. Changes along the line suggested by Dwight Hill of Bell Labs should be considered.41 42

Concerning point 5, in both the EIA review and this one, it was felt to be an area where Ada as well as any other Algol-like HOL (e.g. JOVIAL) could use support. Further, given the complexity of the present support for HOLs, it was worth serious consideration. Although useful support for Ada, some of the changes mentioned in point 5 are inappropriate at this stage of the design.

---

40 There are several optimization techniques that can help with run-time checks. The success of the various techniques depends, in part, on programming style.

41 Hill, D., "A Hardware Mechanism for Supporting Range Checks," ACM SIGARCH Computer Architecture News, Vol. 9, No. 4, June, 1981.

42 Hill's suggestion might be modified by additional instructions which check only one bound. If software optimization showed that one bound could not be violated, the hardware would not need to check it.

Support for uplevel references should be provided. There are many reasons why access to suspended contexts in the context stack is needed. Also, the concept of requiring recompilation of enclosing units when a subunit is recompiled is not really in keeping with the spirit of Ada. Even allowing that it was, cases would arise where recompilation is either impractical or impossible. The compiler would have to have the ability to create a copy of all potentially uplevel addressable parameters on the data stack. The likely result of this requirement is that compilers will not use the Nebula parameter passing mechanism.

WRITING PORTABLE PROGRAMS FOR NEBULA

Portability is the second of the two primary concerns in evaluating Nebula, and is the area containing Nebula's most serious problems. In addition to [PORTABILITY], in which the concepts and deficiencies are discussed at great length, these problems also arise in a number of other papers in Part 2, especially [MISC]. In particular, [MISC] contains the fullest description of the portability issues stemming from the differing Air Force and Army goals, and discusses the overall question of how Nebula's portability deficiencies may impact the Air Force.

"Portability" simply means being able to take a program written and debugged in one environment and have it function identically in another. By definition, portability is at the heart of any standardization effort. It is a major goal of both the DoD and AFSC-HLSS standardization programs.

Both IBM and DEC have produced series of computers that support portability, and it is a significant advantage for each company. As detailed in [PORTABILITY], however, MIL-STD-1862A is not precise enough to be the specification for such a series of computers. Neither the general outline of the machine's operation nor its behavior in an enormous range of special situations is sufficiently well specified for the construction of "plug-compatible" machines by several different vendors. It is possible, in fact, that a single vendor might produce incompatible versions of Nebula, all of which fully satisfy MIL-STD-1862A.

In addition to documenting "some of the most important, most obvious, and most interesting" portability deficiencies in MIL-STD-1862A, [PORTABILITY] also suggests a plan by which a standard architecture may be designed and managed for "controlled non-portability". This requires, first, a complete and unambiguous definition of the ISA, and then a "consistent set of principles" by which implementation dependencies may be permitted. Most important of these principles should be a low cost-benefits ratio, where "cost" relates to the problems caused by non-portability and the difficulty of dealing with them, and "benefits" measures the expected improvement in machine price or performance resulting from allowing the implementation dependency.

PROBLEM AREAS AND SUGGESTED CHANGES

The reports in Part 2 raise many concerns with Nebula, and suggest several changes to the architecture to alleviate these problems. The suggested changes are either in the form of a specific change or of an approach to resolving the particular problem. This section summarizes several of those concerns and suggested changes. It is organized by subsection of MIL-STD-1862A rather than by area of the individual reports. See the appropiate reports in Part 2 for a full explanation of and rationale for the changes.

Procedure Interface

The procedure interface (the procedure call mechanism, the context stack, and the parameter addressing modes) is a central feature of Nebula. However, a majority of the reports discuss problems with and suggest changes to the procedure interface [ADA], [JOVIAL], [PORTABILITY], [VHLL], [VIRTUALIZABILITY], [MISC].

A central feature of the procedure interface is the context stack. A context stack is a stack of procedure contexts that are pushed and popped by the procedure call and return instructions. Each procedure context consists of: the registers used by the procedure, the parameters of the procedure, the PSW of the procedure, and the state of the procedure's exception handler.

The procedure interface provides HOL support by saving and restoring registers and by establishing and accessing parameters. It provides additional support for Ada by maintaining the state of the exception

handler and propagating exceptions. Context switches caused by interrupts, supervisor handled exceptions, debugging breakpoints, traps, and task initiations use the procedure interface. Nebula's support for security is also related to the procedure interface. A context stack can simultaneously contain procedure contexts of procedures executing in privileged mode, in supervisor mode, in both, or in neither.

Allowed access to the context stack: Access to the context stack in Nebula is restricted -- partly to give freedom to implementors. In the current procedure context, the parameters, registers, and exception handler may be accessed via either the addressing modes or the exception handling instructions. Except for the condition code bits and the Enable Arithmetic Error (EAE) bit, there is no way to access the PSW of the current procedure. Access to suspended procedure contexts is very restricted. The privileged instructions, LPSW and SPSW, provide access to the PSW of the caller's context. The only other method for obtaining access to information in suspended context stacks is using the parameters of the current procedure context. These parameters can reference anything that was accessible to the caller which includes the registers of the caller and the objects referenced by the caller's parameters.

Design Principle Violations: Nebula's problems with the procedure interface primarily arise from failing to heed three of its design principles: low risk design; orthogonality; primitives rather than solutions. The procedure interface is actually one of the highest risk fea-

tures of the Nebula design, but there simply wasn't enough time in the design schedule to do the necessary software and hardware evaluation to get the details of the procedure interface correct. A violation of the second principle is the entanlgement of security features with the features supporting HOLs. This entanglement is partly responsible for the difficulty in finding changes to alleviate the problems with the procedure interface.

It is the violation of the third principle that is most obvious; simply being problematic solution, inadequacies with the solution arise -- precisely the reason to avoid solutions and provide primitives. This can be illustrated with two examples of problems associated with providing solutions rather than primitives.43

First, Nebula doesn't provide primitives for helping the compiler writer generate an efficient procedure interface but attempts to provide a solution. The procedure interface provides HOL support by maintaining some of the information traditionally kept on the software-maintained data stack (i.e. the parameters, saved registers, exception handler state) on a hardware-maintained stack (the context stack). The information is used in a particular way: parameters are passed by reference, registers are not inherited (except R1), all registers are always saved and restored across procedure calls. Nebula doesn't help the compiler writer with the data stack maintenance, but rather appropriates part of the data stack, isolates this part from the compiler writer, keeps it on

---

43 Wulf, Willian A., "Compilers and Computer Architecture", Computer, Vol. 14, No. 7, July, 1981, p. 43.

a separate stack (complicating stack space management), and tries to provide the support needed for this stack.

Second, in restricting themselves to a solution, the designers had to trade-off the needs of context switches for interrupts (fast, but with complete state saved) with the needs of HOL procedure calls (putting some operations of the prolog software into the hardware for faster procedure calls, incorporating a flexible approach to amount of state saved, and performing these operations in a manner compatible with various HOLs). It may be that much of the mechanism provided makes more sense as a context switch mechanism and that instead of associating the interrupt and trap invocation mechanism with the procedure interface, it should be associated with task switching.

Procedure Interface problems raised by Reviewers:

References to data in stacked procedure contexts: Since the registers and parameters are objects under the compiler's control, the compiler may need access to them. Not allowing this access restricts what can be stored on the context stack. Earlier reports on Nebula and Ada,[44] [45] [46] have pointed out that this means parameters which are potentially up-level referenced can't exist only on the context stack.

---

44 Davis, M., Nebula as a Target for Ada, IR #655 Intermetrics, February, 1981.

45 Martin, E. W., Fischer, H., Fagg, A., Arnold, R. D., James, J., "Report of MIL-STD-1862 REVIEW COMMITTEE d.b.a. NEBULA REVIEW COMMITTEE", Electronic Industries Association G-33 Data and Configuation Management Committee Computer Resources Task Group, June 1981.

46 Arnold, R. D., Ada and the Nebula Architecture, Boeing Aerospace Corp., February, 1981.

This is also a problem for LISP and JOVIAL [VHLL], [JOVIAL]. It is a problem for LISP not only because of up-level parameter references, but also because of the demands of garbage collection and interactive debugging. Further, though possible in Ada and JOVIAL, in LISP it is impossible for the compiler to determine which parameters are actually up-level referenced.47 Partly because of this, it is felt that implementations of LISP would not use the CALL instruction.

In running virtual machines, privileged instructions must be simulated by the real supervisor [VIRTUALIZABILITY]. To simulate these instructions, the privileged instruction trap handler will need access to the parameters and registers of the procedure context which caused the trap. However, they are inaccessible in Nebula.

Two proposals are presented which give the programmer more access to information in suspended procedure contexts [JOVIAL], [VHLL].

Parameter passing methods: [JOVIAL] and [ADA] also discuss problems with passing parameters. In Nebula parameters are passed by reference; in both JOVIAL and Ada other methods are also required, e.g. call by value. Given this mismatch, parameters which are not passed by reference must be handled separately; they must be explicitly copied by software in some manner [ADA p. 6].48

---

47 To do this analysis in Ada requires recompilation of enclosing units when a subunit is changed.

48 Davis, M., Nebula as a Target for Ada, IR #655 Intermetrics, February, 1981, p. 8.

Hardware solutions to this problem, involving a bit vector in the procedure descriptor to indicate the passing method, are discussed in [JOVIAL].49 Unfortunately, these solutions are not conceptually clean; they add complexity to the procedure interface. Also, rather than resolving the general problem, they only solve it for Ada and JOVIAL (and other languages that use a subset of the parameter passing methods of Ada and JOVIAL).

Procedure Interface and Ada Tasking: Nebula provides no support for the type of tasking provided in Ada.50 Further, by enforcing use of a true stack for the context stack in an Ada program, Nebula's procedure interface provides an obstacle to implementing Ada tasks. To meet the requirements of managing the activation records for a large number of tasks, a more complex data structure, typically a "cactus stack", is required [ADA p. 2]. While a cactus stack can be used for the data stack, the context stack which contains some of the information maintained in the activation records is maintained by the hardware as a simple stack.

To resolve the problems of the context stack supporting Ada tasking, three instructions are proposed: a new internal context switch instruction, a modified CALL instruction, and a modified RET instruction [ADA p.14]. The old CALL and RET instructions would be upward compatible to

---

49 Arnold, R. D., Ada and the Nebula Architecture, Boeing Aerospace Corp., February, 1981, p. 8.

50 Nebula's tasking instructions switch virtual memory spaces and, while useful for scheduling users, are not appropriate for supporting tasking within an Ada program.

the new CALL and RET instructions. The internal context switch instruction would be used for switching Ada tasks; it would change the value of the context stack pointer, but would leave the address space unchanged. These instructions would allow the hardware supported context stack to be a "cactus stack."

Support for the Data Stack: An additional change suggested for Ada support is the extension of the procedure interface to include support for the data stack [ADA p. 17]. This change would not be upward compatible. Other suggestions for this type of support were also made in reviews of Nebula [JOVIAL].[51] [52] If hardware support for high order Algol-like languages is a goal of the architecture, then rather than implementing a portion of the data stack in the hardware, a more appropriate solution is to provide a general support mechanism for the data stack which includes addressing modes for both local and global data and activation frame space allocation.

Effects of Procedure Interface Problems: Without changes to the architecture, the likely result of the mismatches is that compilers will avoid the hardware features in Nebula for HOL support. The above mentioned problems with the parameters may cause compiler writers to avoid the hardware parameter passing mechanism in favor of using the usual

---

51 Martin, E. W., Fischer, H., Fagg, A., Arnold, R. D., James, J., "Report of MIL-STD-1862 REVIEW COMMITTEE d.b.a. NEBULA REVIEW COMMITTEE", Electronic Industries Association G-33 Data and Configuation Management Committee Computer Resources Task Group, June 1981.

52 Arnold, R. D., Ada and the Nebula Architecture, Boeing Aerospace Corp., February, 1981.

software methods. The problems with supporting tasking might cause them to abandon altogether the procedure call and return instructions (CALL, RET) in favor of the "jump to subroutine" and "return from subroutine" instructions (JSR, RSR) [ADA p. 11]. It might be noted that even trying to avoid using the procedure interface can lead to problems as the rest of the architecture (SVCs, OPEXs, etc.) will continue to use it [ADA p. 11].

Traps, Exceptions, and Interrupts

Most of the problems with these parts of Nebula were: associated with the procedure interface (traps, exceptions, and interrupts are invoked through the procedure interface), a result of underspecification, or involved portability issues. The problems related to the procedure interface generally involved lack of access to needed information on the context stack. For discussion of issues affecting portability or resulting from underspecification see [PORTABILITY] and [MISC]. Other issues are discussed below.

Privileged instruction trap: [VIRTUALIZABILITY, p. 5] discusses the problems when a virtual machine, running in virtual privileged mode, tries to execute a privileged instruction. As virtual machines always run in unprivileged mode on the real machine, when the virtual machine "executes" a privileged instruction a privileged instruction trap will occur. The real machine will simulate the instruction for the virtual machine. This presents several problems.

1. Any registers or parameters accessed by the instruction being simulated are in the context stack of the virtual machine and inaccessible to the privileged instruction trap handler.

2. Since aliasing of segments isn't allowed, if the privileged instruction is in an "Instruction Access Only" segment, the segment will have to be remapped with the REPENT instruction before the trap handler may access the instruction.

3. After simulation, the virtual machine must be resumed at the instruction following the privileged instruction which was just simulated. Nebula provides no way for the trap handler to change the program counter of the virtual machine or resume the suspended task at a point other than the point of interruption.

4. Certain instructions, for example PTASK, PRAISE, PINIT, and SETSEG, manipulate or create implementation dependent data and are impossible to simulate without more information than is provided in MIL-STD-1862A.

These points are not only problems for virtual machines. Even when not running virtual machines, the privileged instruction trap handler, the memory management trap handler, the scheduler, and the supervisor exception handler may face some of these problems in trying to perform their functions.

There is also a problem when a virtual machine running in virtual non-privileged mode encounters a privileged instruction [VIRTUALIZABILITY p. 6]. Nebula will trap to the real machine's privileged instruction trap handler. The real machine should then simulate a trap to the virtual machine's privileged instruction trap handler. A task can be initiated on the virtual machine's virtual kernel context stack at the address specified in its trap vector using the tasking instructions, but Nebula doesn't provide a mechanism for the required parameter to be passed.

One of the changes suggested to help make the ISA virtualizable is the addition of a PRIVEX vector that would function analogously to the OPEX vector. [VIRTUALIZABILITY p. 12]

Exceptions: [JOVIAL p. 46] recommends reserving an exception code which the JOVIAL compiler would use for ABORT statements. This would enhance portability of JOVIAL programs. It is further suggested that some exception codes be reserved for hardware exceptions that may be added in future revisions of Nebula [MISC]. These points suggest that to enhance portability, exception codes should be treated like unused opcodes and be either reserved or allocated to particular groups.

In Nebula, propagation of an exception isn't required to be interruptible. Thus a long call chain could lock out interrupts for significant periods of time -- potentially causing problems for both security and system responsiveness.

Interrupts: [FAULT-TOLERANCE] and [I/O] commented on interrupts in Nebula. The only problem raised by [FAULT-TOLERANCE] was the absence of a concept of processor failure. [FAULT-TOLERANCE] suggested the ability to have an "I am dead" signal so the hardware could signal the software and/or the outside world of a processor failure, and a "you are dead" line so a processor, upon detecting failure of another processor, could signal it to shut down. To effect this, a processor failure interrupt should be added which would have as a parameter an implementation dependent fault code indicating the type of failure. The interrupt handler could be part of the non-portable kernel software associated with a particular Nebula implementation.

Because of the specification of the location of the interrupt vectors, a maximum of ten I/O controllers may be attached to a Nebula implementation. If directly connected devices which can interrupt Nebula are also attached, this number is even smaller [I/O pp. 15, 19]. Another problem is that the association of physical vector addresses for the interrupt and the address of the IOC register block causing the interrupt is not programmer visible. [I/O p. 19] suggests changes to the architecture to solve these two problems.

Memory management

The memory management system wasn't directly studied as part of this evaluation (see comments under "Areas Needing Further Study" below). However, [VHLL], [VIRTUALIZABILITY], [PORTABILITY], [MISC], and [DATABASE] did comment on the memory management system.

The memory management system was felt to be inadequate for large LISP programs [VHLL p. 6]. A scheme to use the memory map to implement a demand paging system was discussed. However, subsequent changes to the standard have made any such schemes impossible.

[VIRTUALIZABILITY] suggests that the restrictions on the number of segments provided be removed. It suggests that there should be no restriction, but rather a performance penalty if the number of segments in a map exceeds the number of cached entries provided by the hardware.

There is also a problem with supporting the number of segments that exist in the real machine on the virtual machine [VIRTUALIZABILITY p. 3]. When executing in the virtual machine, one of the segments in the supervisor map will have to be reserved for the trap and interrupt handlers of the real machine. This will deny the virtual machine of at least one segment in its supervisor map.

I/O

Several papers commented on I/O in Nebula [MULTIPROCESSING], [I/O], [VIRTUALIZABILITY], [DATABASE]. For both multiprocessing and database application, it was important to be able to offload communications processing to the IOCs. [MULTIPROCESSING] found the IOC inadequte for this task. Timer support was required in the IOCs. More instructions were desired to perform checksum calculations and bit stuffing operations.

[I/O] commented on several problems in the area of the IOC to CPU interface.

1. The SETSEG description contains no restriction on issuing

the instruction while the IOC is active.

2. The maximum IOC interrupt priority is not normally program visible; the priority of an interrupt is coded in the INT instruction which resides in a instruction-access segment. Therefore, a program cannot find out the maximum priority allowed by an IOC, and would have trouble changing the priority in the INT instruction even if access were allowed. [I/O] suggests that requests for greater than the maximum IOC priority be treated as if they requested the maximum and no IOC error be reported.
3. The state of a Nebula machine following a RESET or IPL sequence is inadequately specified. Also, the format of the IPL data should appear in the standard.
4. The limitations on the number of IOC's (10) may be severely limiting in larger configurations. [I/O] suggests solving that problem along with problems caused by lack of program visibility to the interrupt vector by treating IOC interrupts in a fashion that is consistent with SVC and OPEX interrupt handling.

Instructions

String Instructions: [DATABASE p. 6] proposed modifying the CMPBK (Compare Block) instruction to test for all six relational operators (=, <=,

>=, <, >, <>), rather than only equality; this report also proposed modifying CMPBK to work with strings of different lengths. [PORTABILITY] raises several issues concerning interruptible instructions.

Procedure call and task switching instructions: As discussed above, [ADA p. 14] proposes modifying the context stack to support tasking in Ada. This involves upward compatible changes to the CALL and RET instructions and the introduction of a new internal context switch instruction. This instruction would change the task context stack pointer but would not change the user map pointer. Thus a task switch would occur but the same address space would be used. An additional, but incompatible change is also suggested to the CALL and RET instructions -- support for allocation of the data frame. The above mentioned support for the allocation of context frames would be extended to support data frames.

[VIRTUALIZABILITY pp. 7, 11] discusses the need for the real machine to be notified whenever the virtual machine changes privilege or supervisor/task status. That is, the real machine must know in what mode, privileged or non-privileged, the virtual machine thinks it is operating. Nebula doesn't allow for this. In particular, ERET, ERP, CALLU, and RET can change this status and the real machine has no way of knowing. Two possible methods of resolving this problem are discussed: addition of microcode to keep track of which mode a virtual machine was operating in, or the addition of a PRIVEX vector (see above, "Privileged Instruction Trap)." An alternative solution which requires more exten-

sive microcode changes is to add an instruction which operates like LTASK but additionally maintains several flags indicating virtual machine privilege status which can then be maintained by the microcode implementing sensitive instructions. This additional microcode for these instructions would not be executed unless a flag was set indicating that a virtual machine was executing.

AREAS REQUIRING FURTHER STUDY

There are several areas of Nebula which were not studied and which need to be investigated, namely:

1. The memory management system has been one of the most controversial parts of Nebula because of: its segmentation approach, its failure to guarantee to the programmer more than 16 segments in a map, and the absence of support for demand paging. It has been accused of biasing the design to constraints of the microcomputer implementations and providing too little support for minicomputers. At the same time, the microcomputer implementations may not require the costly segmentation and relocation hardware. Several of the reports comment on the memory management system, namely: virtualizability, multiprocessing, database management systems and miscellaneous. However, it has not been directly investigated.
2. Both the database management system and multiprocessing

reports stressed the need for the IOC to do some of the communications processing. Since those papers did point out some weaknesses, the ability of the IOC to do this work needs to be further investigated.

3. One of the costliest features in executing Ada programs may be run-time checking for constraint errors. Inefficiencies in program execution speeds resulting from this leads to suppression of the checks. Nebula would serve Ada well by providing as much support as possible for range checking. Presently this is supported only by the RANGE instruction.

4. Several features of Nebula, for example the memory management system and the context stack, require special implementation techniques to be efficient. Investigation of implementation techniques available across the desired performance range, and the effect that possible implementation techniques may have on the reviews of that feature of Nebula needs to be performed.

PART 2

Reports by Reviewers

Section 1

INTRODUCTION

The major effort involved in this review of the Nebula ISA was performing studies in several areas. This section contains the reports resulting from those studies. The areas investigated were:

* Support for Ada
* Portability
* I/O
* Support for JOVIAL
* Fault-tolerance
* Virtualizability
* Support for Very High Level Languages, e. g. LISP and SAIL
* Data-base system support
* Multiprocessing
* Miscellaneous

These are listed in order of priority. The priority of the report affects the findings since a higher priority area will be more justified in suggesting changes to improve support for that area than a lower priority area. Thus, for example, given a construct that hinders both Ada and JOVIAL implementations, the Ada paper is more likely to suggest that a change be made.

ACKNOWLEDGEMENTS

This report has been prepared under subcontract 006723 to Digicomp Research Corporation of Ithaca, New York, for the U. S. Air Force's Rome Air Development Center. David Trad or RADC is the contracting officer's technical representative, and Jim Elkins, of Digicomp, is the subcontract monitor.

This work derives in part from the author's participation on the MIL-STD 1862 Review Committee, a subgroup under the G-33 Computer Resources, Data, and Configuration Management Committee of the Electronics Industries Association. The author would like to thank the EIA and the committee organizers for providing a forum in which issues relating to major new architecture could be effectively aired. Special thanks is due to committee co-chairman Dr. Edith Martin for her work in directing the committee and encouraging constructive review of the MIL-STD 1862 Architecture.

The views and conclusions contained in this report are those of the author, and should not be interpreted as representing the official policies, either expressed or implied, of the U. S. Air Force or Department of Defense, unless so designated by other documentation.

1.0 Introduction

In the industry review of the proposed MIL-STD 1862 computer architecture conducted by the Electronics Industries Association (EIA), a number of potential issues were raised. These issues were classified as management issues, Ada issues, and detailed instruction set architecture (ISA) issues.

During the review process and in a one week intensive meeting of the Nebula Technical Advisory Board in May, most of the issues raised by the EIA's review committee were satisfactorily resolved in one manner or other. An exception, however, was virtually the entire set of issues related to the implementation of Ada on the Nebula architecture. The board reportedly discussed the issues at some length, but could reach no consensus as to a desirable set of changes that would resolve the issues. As a result, it recommended no change to the architecture in the areas in question.

The decision to recommend no changes relating to Ada issues is disappointing, but understandable. The development schedule for the Army's Military Computer Family program (MCF), which depends on the Nebula architecture, created a good deal of pressure to freeze the definition of the architecture early, and to discourage major changes. Resolution of the most serious Ada issues would have required relatively far reaching changes in the architecture. There simply was not sufficient time to develop agreement on such a set of changes. However, the unresolved issues leave DOD procurement officials and prospective implementors of Ada with some difficult questions to resolve.

The most basic question that must be decided is whether to try to live with the standard as it now exists, or to press for at least some additional changes that would facilitate Ada implementations. The sections that follow discuss some of the options that are available, and some of the pros and cons of the various choices. Section 2 briefly reviews the issue of memory management in an Ada multitasking environment, which was identified during the industry review as the single most critical Ada issue. Section 3 reviews other significant Ada issues. Section 4 considers options available if no changes in the standard are sought, while section 5 considers options available when varying degrees of change to the standard are allowed.

2.0 The Memory Management Issue

Ada's concept of tasking is based on the notion that programs can be and often are most naturally organized as collections of parallel, communicating processes. This approach is not new. It has been around for many years in such languages as Simula, Algol-68, Modula, Small Talk, and Concurrent Pascal [5, 14]. It is closely akin to the concept of co-routines, which have been implemented in a wide variety of general and special purpose languages [9, 13]. However, it has not previously been a central feature of a "mainstream" programming language.

A common characteristic of languages employing multiple parallel processes is that individual procedures do not execute in a strictly "last in, first out" manner. This means that a simple stack is no longer a suitable mechanism for managing their activation records. For example, one task, task "A", may execute for a period of time and then be suspended. Another task, task "B", may then execute for a time before it, too, is suspended. If task "A" is resumed at that point and attempts to make a call, it will find that the stack is blocked by activation records for the suspended task "B" which must not be overstored.

2.1 General Approaches

There are basically two solutions to the problem described above. One is to use multiple stacks, reserving blocks of memory sufficient to give each task its own stack. This is an efficient solution when the number of tasks is small and there is limited communication among them. It breaks down, however, when the number of tasks is large. This will be discussed in more detail shortly.

The other solution, and the one which is normally preferred for the implementation of multitasking languages, is to use a more complex data structure than a stack for the management of activation records. One alternative is a fully general data heap. This is expensive, however, unless dynamic memory management is directly supported in the processor architecture. It is also more general than really needed, since procedure activations remain "last in, first out" within individual tasks. A better alternative is therefore a modified type of stack capable of spawning "side stacks" at arbitrary points. Linear storage is used for both the original stack and its offshoots, making the physical structure in memory a sequence of discrete stack segments connected by links. The logical structure, however, is suggestive of a saguaro cactus, so the structure is usually referred to as a "cactus stack". A set of algorithms for the management of a cactus stack is described in appendix A.

2.2 The Nebula Model

For the Nebula model of tasking, the memory management solution of separate stacks is strongly assumed. The Nebula call mechanism divides the general concept of "activation record" into two parts-- a "context frame", managed by hardware, and a "data frame", managed by software. For the allocation of context records, at least, the assumption of separate stacks is firmly "wired in" to the call mechanism. A task must have an open context stack for the call mechanism to function properly, since a new context frame is created simply by decrementing the current context frame pointer by an appropriate amount. On a return, the caller's context frame is reestablished by incrementing the pointer by the size of the current frame. There is no provision for a dynamic link within the context frame, so frames must be contiguous. This minimizes procedure call overhead and optimizes the design for an expected style of programming in which tasks are relatively isolated, and task calls are greatly outnumbered by procedure calls.

Minimizing procedure call overhead is certainly a reasonable goal, but the assumptions behind the Nebula model for tasking are questionable. In particular, the assumption that each task can be allocated a sufficiently large block of memory to run with its own separate stack is out of line with the number of simultaneously active tasks that can be reasonably anticipated for at least some Ada programs. Part of the problem is that the compiler cannot know, in general, how much stack space a particular task will require; depth of recursion is often data dependent, and separate compilations make it difficult even to determine if a task will use recussion. But even if a perfect projection of stack space requirements could be made for each task, memory utilization would still be unacceptably poor in many cases. To avoid memory faults, the compiler would have to allocate to each task the maximum amount of space it would ever need; yet in applications where tasking is heavily utilized, most tasks spend most of their time in a "dormant" state with minimal space requirements.

As an example of how tasking might be used in real time applications, the ACM SIGPLAN Rationale for the Design of the Ada Programming Language contains a sample Ada package for supporting radar track management [11]. The package declares a family of tasks 512 elements deep; each member of this family includes storage for one track, and entry points to initialize, read, or change the track information. The structure of the task insures that a track can be initialized only once, and that it cannot be changed while being read. Within a larger radar surveillance program, the entries of these tasks would be called very frequently. Such calls might, in fact, outnumber ordinary procedure calls. It is difficult to imagine how the Nebula model of tasking could work for a program with so many tasks and such frequent task calls.

The radar track management package described in [11] is simplified for purposes of illustration. However, it is not a "contrived" example, unlikely to arise in "real world" applications. It would admittedly be possible to write a track management package which did not employ such extensive use of tasking, and which would be more suitable for implementation on Nebula under its current tasking model. However, the package as described makes an entirely reasonable use of tasking. It was included in the SIGPLAN document specifically to illustrate the software engineering advantages of this feature of Ada. It typifies an approach to programming which has been used successfully and to considerable advantage in languages like Simula. It is unreasonable to argue, then, that Nebula should not be required to support this type of usage.

3.0 Other Issues

3.1 Call Mechanism Performance

During an industry review of the Nebula architecture conducted under the auspices of the Electronics Industries Association, the procedure call mechanism was the focus of much criticism and comment. The bulk of reviewers' responses could be divided more or less equally into two categories. Some reviewers felt that the mechanism did too much in hardware, precluding software optimization of register usage and calling sequences. Others felt that it did too little by failing to allocate a data frame for the called procedure.

Regarding the first point, there certainly are implementation approaches which might give poorer performance for the hardware based call mechanism of the standard than for an optimized software mechanism. An example would be one which simply maintained users' registers and other context information internally during procedure execution, waiting until the next procedure call to store it in the context region of memory. This approach would generate essentially the same memory references as a simple-minded software mechanism; conceivably, a smarter software mechanism which allowed less than the full set of registers used by a procedure to be saved would give better performance. However, there are other implementation approaches which would give much better performance than any software based mechanism.

An example of an implementation giving good performance in the call mechanism is one using parallel write-through from the context cache to the context area of memory. In that case, no time is required to save the caller's context when making a procedure call. Alternatively, a mechanism able to maintain multiple procedure contexts within its cache would also perform well. The point is that the architecture does not prescribe the implementation and is not intended to. Criticism based on performance considerations for one particular type of implementation is therefore largely irrelevant. The definition of the call mechanism allows designers to select implementations which do give good performance, and in that respect, at least, it makes good architectural sense.

The last statement above is not meant to imply that there are no problems with the Nebula call mechanism. They relate to its usability for the implementation of the Ada language, however, rather than to the performance of the basic mechanism itself. The issue of memory management and the call mechanism's dependence on an open stack was discussed in section 2 above. An additional issue concerns the method in which parameters are passed and referenced.

3.2 Parameter Passing and Referencing

Problems with the parameter passing and referencing provisions of Nebula's call mechanism exist, but appear less serious than those associated with memory management for tasking. As discussed in [7], the software workarounds required to match the Ada semantics are not unreasonably expensive.

The specific issue is that Ada requires scaler parameters to be passed by value, so that programs will be more likely to behave consistently under different multitasking implementations. Except for literals, however, Nebula passes parameters by reference. Ada also requires that parameters of an outer procedure be visible to procedures nested within it. Nebula provides no method to reference parameters outside of the procedure to which they belong.

To achieve parameter passage by value with Nebula, either the calling procedure must copy scaler parameters to local temporaries, passing descriptors of the temporaries as the actual call parameters, or the called procedure must copy parameters to local temporaries on entry, referencing the temporaries within its body rather than the parameters themselves. The former approach will often involve no added overhead to the call, since it is the necessary method to handle output parameters when the value of the parameter within the calling program is constrained. However, it does not solve the problem of uplevel referencing of parameters, for which the latter approach is required.

3.3 Tasking Operations

An item that was identified as a significant unknown during the EIA review of the Nebula architecture was its suitability for the implementation of various Ada task management operations. For instance, would adequate performance of Ada programs require special architectural features to support parameter passing on task calls, selective wait statements, or task exit monitors?

A significant conclusion of the present study is that, with one major qualification, such features do not appear to be needed. It may ultimately prove desirable to extend the Nebula instruction set with operations tailored to the management of tasks, but the nature of the operations required is compatible with implementation under the current OPEX facility. No underlying changes to the basic architecture appear to be required.

The qualification is that, to achieve the type of data sharing and communication between tasks required by the language, Ada tasks must use commonly mapped data spaces. It is not feasible to use separate data spaces for each task and depend on operating system software to manipulate map entries on a case by case basis when sharing is required. This, in turn, means that data frames

cannot be allocated from a simple stack within a task's own space. A more complex algorithm, such as that of Appendix A, must be used to allocate frames from a space that is commonly mapped for all tasks. If the allocation is performed in software, there will be a significant increase in procedure call overhead. With careful implementation of the algorithms of Appendix A, the increase may be considered tolerable; however, it argues in favor of the changes discussed in section 5.3.1 below.

3.4 Error Detection and Diagnosis

Another issue raised in the EIA review is that the architecture's capabilities for trapping software errors, while acceptable by the standards of most current commercial architectures, are weak for an architecture of Nebula's generation and intended lifespan. The Intel 432, for example, provides run time error detection capabilities that dramatically eclipse those of the Nebula architecture.

There is a definite run time penalty associated with the type of architecture used in the 432, and it may be appropriate for Nebula to avoid such an approach. Nevertheless, there are a number of features that could enhance Nebula's capabilities for trapping software errors without impacting its run time performance. In some cases, execution performance would probably be enhanced by eliminating the need for more expensive run time checks implemented in the software. This is a significant issue, since the definition of Ada requires a high level of error checking. Checks which cannot be performed through static analysis of source code and are not performed by the hardware must be performed through run time checks in software.

One of the specific issues in this category is that, under the Nebula architecture, all procedures of a task have common access privilages throughout the addressing space of the task. This does not correspond to the scoping rules of the Ada language. It was pointed out in [4] that display based addressing, with display entries including the size of the associated data frame, would allow a much closer mapping of addressing capabilities to the semantics of the Ada language. It would also simplify compiler design and lead to performance enhancements in generated code. Unfortunately, it is not compatible with the current standard, even as an upward extension. It is probably a more drastic revision than can be justified by the benefits it would provide. For that reason, it is not among the recommendations included in this report.

4.0 Living with the Current Standard

4.1 Non-Solutions

One "solution" to the problem of supporting extensive multitasking on the Nebula architecture in its current form is simply not to do it. Projects could specify the use of a language other than Ada, or of a dialect of Ada that restricts multitasking.

Failure to support full Ada will probably not be acceptable to most program offices. It does, however, have the virtue that it makes clear what is and isn't being done. This is preferable--at least technically, if not politically--to another "solution" that would have the same practical effect, but would conceal it under a veneer of superficial support for the full language.

There are various "brute force" methods of supporting multitasking that are extremely inefficient if tasking is used extensively, but perform well if it isn't. An obvious example would be the use of tasks or processes at the operating system level--i.e., as recognized and managed by conventional multiprogramming operating systems--to model individual tasks within an Ada program. Implicit in this approach is the use of separate virtual address spaces for the individual tasks, and special operating system calls to permit access to shared variables.

It is easy for implementors of such approaches to shrug off poor tasking performance as the penalty for use of "inherently expensive" language features. They can claim that the implementation is optimized for the most commonly encountered programming style--i.e., ordinary procedure calls. Programmers are usually quick to learn which features of a particular implementation are efficient and which are expensive. They adjust their coding technique accordingly, so the claim of optimization for common practice quickly becomes a "self fulfilling prophecy".

It is important that program offices recognize that there is no technical merit to any claims that Ada tasking is impossible to implement efficiently on conventional architectures. Ada tasking is not fundamentally different from that found in languages such as Simula, and efficient implementations for these languages do exist. There are, of course, degrees of efficiency, and the architecture on which a language is implemented can significantly impact its ultimate efficiency. The problem is more acute when an architecture introduces mechanisms, such as the current Nebula call mechanism, based on models that are incompatible with the requirements of a multitasking language. Nonetheless, "reasonable" implementations of tasking remain possible.

4.2 Using the Current Call Mechanism

Assuming that the non-solutions described above have been ruled out, then Ada implementors constrained to work with Nebula in its current form face an "interesting" challenge [*]. They must overcome or circumvent the memory management limitations of the current call mechanism with respect to tasking, without causing excessive overhead for programs not using tasking.

The implementation approach which most nearly conforms to the current Nebula model would rely on the memory management features of the architecture to permit dynamic growth in the context region associated with each task. The memory management scheme is illustrated in figure 4.2-1. It specifies a set of map registers, with each register containing the start address for a segment of virtual address space, an offset for translation to corresponding physical addresses within the segment, and a segment access code. No end address for the segment is required, since each segment is assumed bounded by the start address above--or by the largest virtual address in the case of the last map entry.

Given the memory mapping capabilities, it is possible to initially allocate only a modest amount of physical context memory for each task. The map entry for the region initially allocated is preceded by several entries describing segments of virtual address space marked as inaccessible--no physical memory allocated. If a task overflows its initially allocated context space, it will attempt to access one of the inaccessible regions, generating a memory management trap. At that point, the operating system can allocate a physical memory block to the segment generating the trap. The task can then be resumed with the expanded context address space. The memory map patches together separate regions of physical memory into a contiguous region of virtual address space.

To limit the number of memory faults for tasks with deep recursion, the size of regions allocated should probably double with each successive fault. To avoid depleting the pool of physical memory available for allocation as context memory, vacated segments must be reclaimed by the operating system on each task switch.

The above scheme is usable for the hardware context stack, because there is no need to share such memory between tasks. It is not practical to use it for a data stack, due to the complexities introduced by sharing. Allocation of data frames, in the general case, must depend on some type of software mechanism more complex than a stack, with allocations made from a pool that is common to

---

[*] "May you live in interesting times" is a Chinese curse not usually appreciated by westerners, unless they have been software project managers.

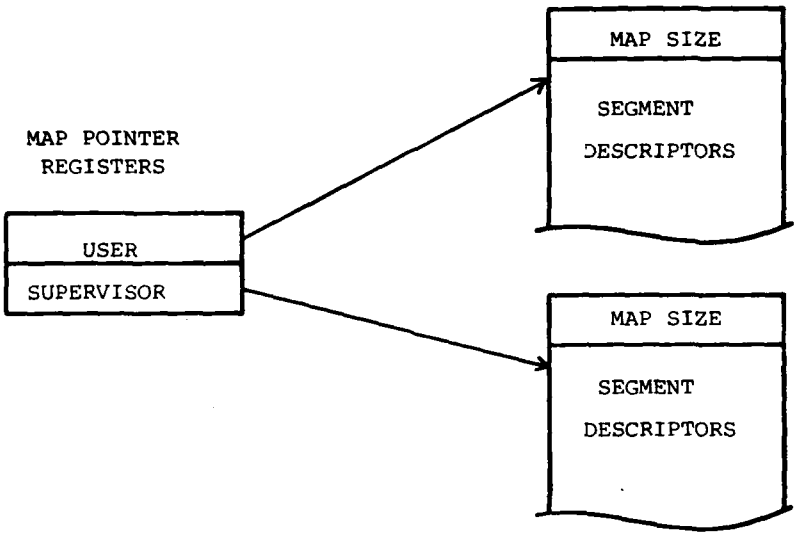


FIGURE 4.2-1a: MAP DATA STRUCTURE

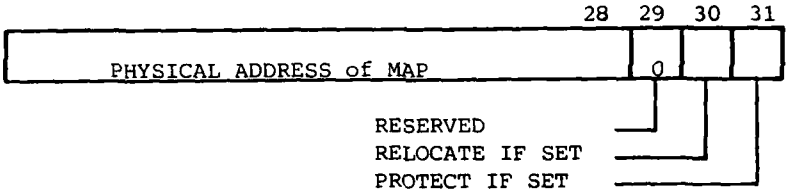


FIGURE 4.2-1b: MEMORY MAP POINTER REGISTERS

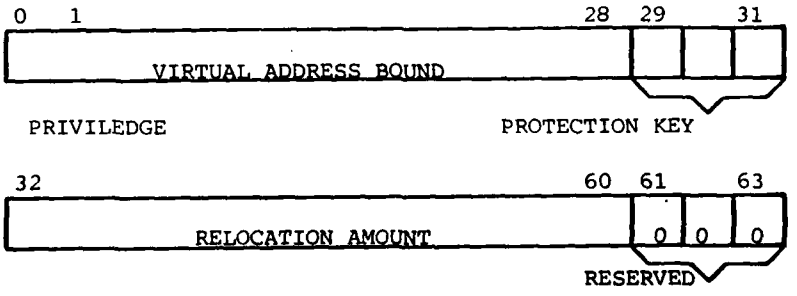


Figure 4.2-1c: Map Entry Format

all tasks.

The main drawback of the approach just described is that it makes task switches relatively expensive. It also makes address translation mandatory, even for embedded software applications which otherwise have no need for it. This defeats a provision of the architecture which would allow such applications to avoid unnecessary overhead by running with address translation disabled.

4.3 Bypassing the Call Mechanism

As an alternative to the approach of the preceding section, it is possible to bypass Nebula's procedure call mechanism. The instruction set includes more primitive call and return instructions called "jump to subroutine" (JSR), and "return from subroutine" (RSR). These instructions do not cause the allocation or freeing of a hardware context frame, and can be used as the basis for a "conventional" software implementation of tasking.

With a software based call mechanism, procedure activation records are allocated following an appropriate algorithm, and call parameters are passed by storing them in a reserved portion of the activation record. Since a new context frame is not being created, registers are inherited across the procedure call and may need to be explicitly saved.

Activation records for a given task are linked through a dynamic link field in the activation record; task switches are accomplished by saving state information for the current task and loading it for the new task. State information includes, in the general case, working registers, program counter, PSW, and frame pointer. However, if the task switch is the result of executing an Ada task call or accept statement, working registers and PSW can be ignored. The compiler simply observes the convention that working registers and condition codes are not preserved across task calls, and generates code accordingly. This minimizes task switch time for the most common type of switching.

The above approach reflects the way multitasking is typically implemented on conventional architectures for languages like Simula and Modula. It would seem to be safe for Nebula as well. However, even this approach encounters certain complexities. Since use of the formal call mechanism is implicit in the extended opcode (OPEX) and in the executive service call (SVC) facilities, then some care is needed in how tasks are allowed to use these facilities. If an internal context switch can occur as a result of an SVC or as a result of an interrupt during execution of a software implemented OPEX, then the kernel software must include some provision for managing the open context frame as part of the internal context switch. Details of this process need to be worked out, although the process itself would not appear to be critical to the run time performance of the implementation.

4.4 Compiler Issues

An alternative approach that will be attractive to many implementors will be to devise compiler optimization strategies which, coupled with appropriate run time storage management strategies, could salvage the current call mechanism. For instance, while it is not possible to allocate sufficient memory for a separate stack to each of 500 or more simultaneously active tasks, it is certainly possible to do so for half a dozen or so. If the bulk of the tasks, such as those associated with active data structures, can be transformed into "monitors", as described in [8], then a half dozen or so stack spaces may be sufficient to implement the program in question.

Even if there are too many tasks remaining after "monitorizing" to allow assignment of a separate stack space to each, it might be possible to treat a limited number of stack spaces as resources to be dynamically shared among the active tasks. Inactive context stacks could be stored in dynamic memory, packed together. To execute, a task's context stack would first have to be copied into a region of open stack space. But because there would be a number of open stack spaces available, the most active tasks would normally retain their stack space between periods of execution, and most context switches could occur without the need for stack copying.

It is difficult to estimate how far the process of "optimizing away" tasks can actually be carried. In principle, it should be possible to write a "program sequentializer" that would transform any program expressed as a set of parallel processes into an equivalent process involving a single thread of control. This involves the same kind of tradeoffs that are involved in compiled vs. interpreted code. More work is done at compile time to save overhead at run time, but at the cost of considerably greater software complexity overall. There also tends to be a loss of flexibility in making changes and greater difficulty in program debugging.

The problem with depending on compiler optimization schemes and clever run time storage management strategies for implementation of multitasking on a single processor is that it represents largely unexplored software territory, and hence must be viewed as high risk. The algorithms are bound to be complex, and there is a great deal of room for hidden bugs, even if the theory of the algorithms is soundly developed. It is an area that deserves considerably more study, but in general it seems a lot of work to go to for a result that could more readily achieved by appropriate design of the target architecture.

5.0 Options Involving Changes to the Standard

5.1 Fully Compatible Changes

There are three levels of potential changes to the architecture which might be considered. At the level of minimal changes would be changes which permit both upward and downward software compatability with respect to the current version of the standard. That is, any changes would be limited to new instructions whose effects could be simulated through software using the OPEX mechanism on older implementations of the standard. Properly speaking, these do not constitute actual architectural changes, since the current standard explicitly allows for such extensions.

Unfortunately, the OPEX mechanism is of little use in addressing the most serious Ada implementation issues. It is firmly rooted in the current CALL mechanism, and it is precisely that mechanism that causes trouble for Ada implementations. For instance, suppose an alternate form of the CALL instruction, more suited to the Ada tasking model, were desired. Call it DCALL, for "dynamic call". The purpose of the instruction would be to allow a task to execute from a context frame which had been "boxed in" on the stack by the frames of other tasks. DCALL would detect the boxed-in condition in some manner--say by comparison of the current context pointer to a global stacktop pointer--and obtain storage for the new context frame by locating a new free block.

The DCALL instruction could be implemented in microcode with no great trouble, but it cannot be simulated in software through the OPEX mechanism. The problem is that a software implemented OPEX looks just like a conventional procedure call. In particular, it allocates a new context frame for use by the OPEX software, and that frame is allocated according to the Nebula model of a guaranteed open stack. If the OPEX is executed by a task with a boxed-in context frame, the frame allocated for the OPEX would overstore the frame of another active task.

5.2 Upward Compatible Changes

This leads us to the second level of changes--those preserving upward compatibility with the current standard, but precluding downward compatibility. This set can be arbitrarily large, provided that one of the changes is the incorporation of a hardware "compatibility mode flag" to control interpretation of instructions. This is a legitimate approach, and has been used in DEC's VAX computers to allow compatibility with the architecturally dissimilar PDP-11 family. However, for our purposes here, we will consider only changes which do not require a compatibility mode to achieve upward compatibility.

5.2.1 Call Instruction

The upward compatible changes are based on a slightly modified call mechanism that incorporates a cactus stack algorithm for context frame allocation in place of the current simple stack scheme. The algorithm used is essentially the same as that described in Appendix A. It results in a transparently modified CALL instruction that allows tasks to execute from context frames that are "boxed in" by frames from other tasks. The instruction detects the boxed-in condition, and obtains storage for the new context frame by locating a new free block.

A boxed-in stack is detected by associating with each logical task, at the time it is readied for execution, a "current working block". The current working block is characterized by a block limit address, which is maintained in the CPU and used to determine when there is sufficient space for allocation of a new context frame. This is described in more detail below.

5.2.2 Return Instruction

The modified CALL instruction requires a modified RET instruction to make it usable. The modification is required in order to allow the RET instruction to execute properly when returning to a procedure whose frame was boxed in. In that case, the RET instruction cannot get to the caller's context frame simply by incrementing the context pointer by the size of the current context frame. A dynamic link stored in the current context frame is required.

The dynamic link need not be stored on every CALL or used on every RET. It is needed only when the CALL skips over a blocked region of the stack. There are two bits in the PSW currently reserved for "implementation dependent usage". One of these can be used to signal that a dynamic return is required and thereby control the operation of the RET instruction.

5.2.3 Internal Context Switching

Another concomitant change that is needed for the modified call mechanism is an instruction to perform internal task context switching. "Internal" means that the switch occurs within the framework of what would currently be considered a single Nebula task. The context pointer register is changed, establishing the general registers, parameters, and PSW for a new logical task, but leaving the memory map unaffected.

The internal context switch instruction must interface with memory management firmware to preserve the integrity of memory management data structures. The firmware routines it must invoke correspond to the CLOSE_TASK_STORE and OPEN_TASK_STORE of appen-

dix A. The context switch instruction takes four distinct operands. The first two receive the context pointer and the free list pointer for the currently executing task. The second two furnish corresponding values for the task to which execution is being switched. The second and fourth operands may be the same, but need not be. If they are not the same, then the storage pools used for context memory by the two tasks are isolated from each other. This means that it is possible to allocate a dedicated block of memory for a task that must execute with minimum interference from other tasks, without resorting to creation of a separate Nebula task.

5.2.4 Hardware/Software Interface

The algorithms described in appendix A use a "boundary tag" method for management of the dynamic memory pool. This is convenient for hardware based memory management, in that the state information needed to manage the memory pool is contained in control words within the pool. They are thus equally accessible to call mechanism firmware and operating system software. No special maps need be maintained in memory accessible only to the hardware. Aside from the current context pointer itself, the only information needed for firmware to manage the pool is a pointer to the upper or lower bound of the current working block [*].

The limited context information required for hardware management of the memory pool simplifies the interface between operating system software, which must set up the pools initially and modify them on occasion, and the firmware, which manages the pools during program execution. Although allowing software access to structures that are manipulated by firmware might initially appear dangerous, the control words are protected by the memory access codes of Nebula's memory mapping system. They would reside within memory segments marked as "context only", and could not be corrupted by unprivileged software.

5.2.5 Hardware Implications

There is not much question that a hardware based capability for cactus stack management of context frames would make multitasking more efficient. A central question in evaluating the proposed changes, however, is the degree to which the addition of this capability impacts hardware complexity and the execution speed of normal procedure calls and returns.

With reasonable hardware support, the proposed changes need not

[*] If the stack grows upward, then the pointer is to the upper bound. If it grows downward, then the pointer is to the lower bound.

impact execution speed at all. Nor is their impact on hardware complexity particularly significant. A register containing the limit address of the current working block is required, and a stackpointer register separate from the context pointer register is convenient. The limit register is used to test, after determining the size of a new context frame, whether there is sufficient room for the frame within the current working block. If there is, the action taken is identical to that taken under the current call mechanism. Extra cost is incurred only if the test fails, which corresponds to what would be a memory fault with the current call mechanism.

If needed for highest performance, the test for block overflow can be made to consume zero time in the normal case. The test can be initiated in parallel with the micro instruction sequence appropriate to passing the test. Success of the test would do nothing, while failure would generate an overriding branch in the micrologic. The microcode accessed by the overriding branch might have to undo actions initiated by the default code, but its task in this respect would likely be simpler than it would be under the current mechanism. In the latter case, the overflow is not detected until the firmware attempts to access inaccessible memory, which may be much later in the micro sequence.

Masking additional execution time for the return instruction is even easier than it is for the call instruction. The action appropriate on a return instruction is completely determined by the combination of the dynamic return bit in the PSW and a "history bit" indicating the prior state of the dynamic return bit. These bits can be used to gate or modify the initial control store address for the return instruction. If a normal return is appropriate, it is executed immediately with no overhead for testing or unnecessary actions.

5.3 Incompatible Changes

Given the political complications of changing standards and the inertia that published standards quickly acquire, it is much more comfortable to propose upward compatible changes than to try to justify incompatible ones. In the case of MIL-STD 1862, however, there is little technical reason to limit consideration of changes to those that are upward compatible. Hardware under development is now limited to prototype implementations of the standard, and will remain so for some time. There is no significant accumulation of software developed for the current standard that must be protected. Provided that proposed changes are not so extensive as to invalidate experience gained in implementation of prototypes, they deserve consideration on their technical merits.

Although the upward compatible changes described in section 5.2 above remove the obstructions to use of the Nebula call mechanism for implementing Ada tasking, they fall short of providing a

really efficient call mechanism. In particular, they leave the allocation of procedure data frames to prologue software. This deficiency, not serious in the absence of tasking when a simple stack policy of storage management can be used, becomes more serious in the presence of tasking.

5.3.1 Hardware Allocation of Data Frames

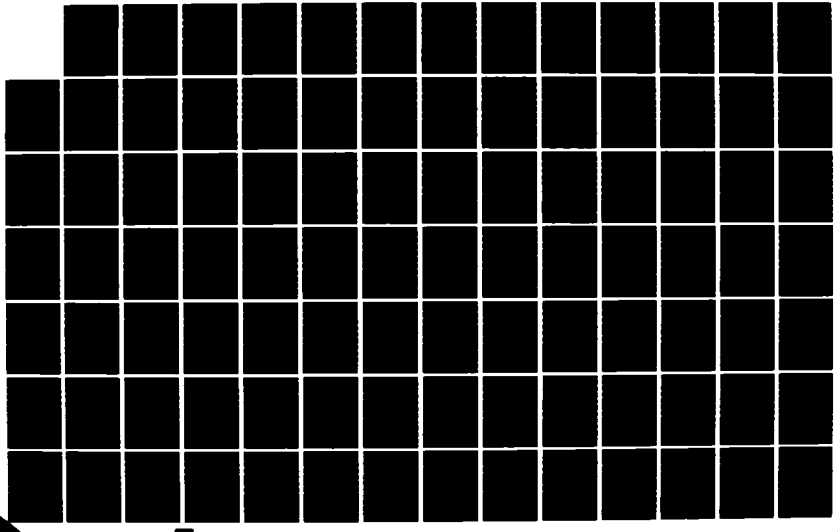
If "cactus stack" allocation of context frames is introduced into the hardware to salvage the basic call mechanism, it would seem reasonable to use it for allocation of the data frames as well. The procedure descriptor, which currently gives the number of parameters and the maximum number of registers used by the procedure, can be amended to include also the size of the basic data frame. This allows the call instruction microcode to allocate the data frame automatically. It also allows call parameters to be stored at the base of the data frame rather than in the context frame, solving the uplevel parameter referencing problem.

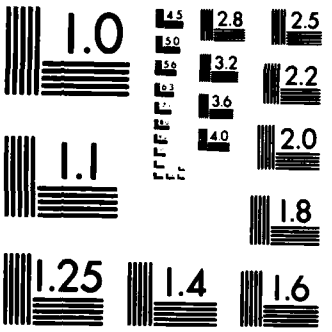
Using hardware based memory management for data frame allocation causes certain complexities that do not arise when it is confined to allocation of hardware context frames. The complications are not serious, in that they have various reasonable solutions, but they do require that solutions be adopted. For example, some mechanism must be provided to return the frame pointer allocated by the call instruction microcode for use by subsequent compiler generated code. One obvious alternative is to provide the call instruction with an additional operand specifying where the frame pointer is to be stored. This would normally be a register, and parameters would be referenced as local data using the byte indexed mode. An alternative to this, suggested by Jim Elkins of Digicomp Research, Inc., is to hold the frame pointer in an internal CPU register until it is accessed by a specific new instruction--"get frame pointer". This is less efficient than the former method, but has the advantage of not requiring changes in the operands of the OPEX and SVC instructions. These also use the call mechanism.

An additional complication has to do with protection of the memory management block control words from user software. This is not a problem when the control words reside in segments used for context memory only, but it is a potentially serious problem when they reside in segments used for general access by user software. There would appear to be only two viable solutions to this problem. One is simply to use a memory management scheme other than the boundary tag method, in which the control structures could reside in protected memory separate from the memory whose allocation they control. There are such algorithms, but none of them appear to be as simple and well suited for the type of usage required as the boundary tag method. A better alternative is discussed below.

AD-A151 041 NEBULA INSTRUCTION SET ARCHITECTURE (ISA) EVALUATION (U) DIGICOMP RESEARCH CORP ITHACA NY R D ARNOLD ET AL. SEP 84 RADC-TR-84-190 F30602-80-C-0279 2/4

UNCLASSIFIED F/G 9/2 NL





5.3.2 Write Protect Word and "Uninitialize"

5.3.2.1 System Usage

A more attractive way to protect dynamic memory control structures would be to provide the capability to write protect individual memory words regardless of the access code of the segment in which they reside. This not only solves the particular problem of protection for dynamic memory control words, but it provides a very useful mechanism of quite general value. It simplifies the development of software debugging tools, and allows them to have capabilities that simply cannot otherwise be implemented. It aids detection and diagnosis of errors in operational software, and maps well onto Ada's concept of private variables.

A similar and complementary capability to write protect by word is hardware detection of uninitialized variable references through the use of an "uninitialize" instruction. This is a particularly useful capability, in that it detects a common and insidious type of software error against which there is no other satisfactory method of protection.

To be most generally useful, the ability to write protect and to set the uninitialized state on a memory variable should be available to user software. At the same time, the capability must be restricted so that user software cannot alter the status of words it does not logically "own". With hardware allocation of data frames, there is a reasonably straightforward way to do this. The hardware knows the location and size of the current procedure's data frame, and can restrict alteration of protection and initialization bits in the user mode to the appropriate segment of memory. The assumption here is that a procedure or function "owns" exactly that memory located within its local data frame, which seems reasonably close to the intent of Ada's semantics.

There is one qualification on the last statement above. Ada defines objects of access variables to be allocated from heap storage independent of a subprogram's local data frame. To allow user software to write protect or set uninitialized such objects, it might be desirable to provide a new segment access code that would allow unrestricted alteration of word write protect and initialization status. Several reserved access codes are currently available, so such a function is feasible. As a refinement, the combination of "uninitialized, write protected" on a word could be interpreted to prohibit user alteration of its status as well as normal read/write access. This would allow embedding of hardware or operating system protected variables within a segment with otherwise unrestricted user permissions, which is useful for implementing a secure, general purpose data heap.

5.3.2.2 Hardware Impact

Detection of uninitialized variable references is relatively easy to implement. It amounts, mainly, to an expansion of machine word size by one bit. When a word is written, the "uninitialized" bit is cleared. When it is read, the bit is tested, with a trap generated if the bit is not cleared. There is a special instruction which sets the bit in a specified word or block of words.

Write protection by word, on the other hand, is sometimes considered an expensive feature to implement in current generation machines. If the most straightforward implementation approach is followed-- i.e., reading the word and testing its write protect bit in the CPU prior to writing--memory bandwidth on write operations is effectively halved [*]. However, there are reasonable alternative approaches which are well suited to current technologies.

Memories are usually organized as independent banks of storage with buffered access to a common bus. By adding a minimal degree of intelligence to the memory to bus interface, it is possible to convert a write operation locally to a read of the appropriate write protect bit, meanwhile latching the write data and address. The status of the write protect bit is returned immediately to the CPU, which then proceeds with its next operation. The latched write data and address are held within the memory bank until there is an idle cycle in which it can be stored, or until the next write to the same bank. If the latter occurs first, the memory bank must signal a hold while it stores the previously latched data. However, with interleaved addressing--low order bits select bank--there is a high probability of a read or write operation to another bank prior to the next write to the original bank. This gives the original bank the opportunity to store the latched data, making delays rare.

With a 32 bit machine, the 6% increase in memory size which the bits for write protection and detection of uninitialized variable references represent is hardly a major consideration. The cost of memory chips is a small part of total hardware cost, and the cost of hardware is a small part of total system costs. It is now routinely eclipsed by the cost of software, and any feature which helps to reduce the cost of software development and to enhance reliability without impacting performance is probably justified.

Introduction of write protect and uninitialized variable bits does raise some design questions which must be answered. What should happen, for instance, when a single byte within an unini-

[*] Ironically, on older core memory machines, this approach cost nothing, since a read before write was essential to the operation of the memory.

tialized word is stored? The obvious answer, that it causes the status of the word to change to initialized, lacks something in aesthetic appeal. It seems to conflict with the notion that bytes are individually accessible. Yet extending initialization status protection to individual bytes seems like unnecessary overkill.

A more substantial question is how initialization and write protect bits should be handled during I/O. For byte stream I/O to and from serial devices, it is probably reasonable to ignore the bits. The convention would be that anything input in this manner would be considered as initialized and not write protected, except as provided by the access code for the segment into which it was read. The real problem is with disk I/O of program and data images. If automatic swapping is envisioned as an adjunct to virtual memory, it would seem essential to use a word oriented disk device that included storage for the write protect and initialization status bits. This is certainly possible, but it is rather unconventional.

Despite these complications, it is the author's conviction that word oriented write protection and initialization status detection would be of major value in reducing software life cycle costs. This is based on extended experience on several large aerospace software projects, covering design, development and maintenance phases. The benefit that could be expected is almost certainly sufficient to justify inclusion of these capabilities in any modern architecture claiming to be oriented toward reduction of software costs.

5.3.3 A Further Note on Compatibility

Strictly speaking, hardware allocation of data frames can be achieved without sacrificing upward compatibility. It is merely necessary to make the new call a separate instruction, rather than a modification of the existing instruction. However, a new return instruction would also be required to complement the call. The Nebula vectored operations (interrupts, traps, service calls, and unimplemented op codes) couldn't use hardware allocation of data frames if upward compatibility were preserved, since there is no convenient way to distinguish new forms of these operations from the existing forms. For these reasons, an outright change in the call mechanism, rather than introduction of new instructions, is probably preferable. It would give a cleaner, more consistent architecture. As noted in section 5.2.3 above, the new forms can be implemented with no execution time penalties even when tasking is not used. In the absence of any significant body of old software to protect, there is no technical reason to retain the older forms.

Appendix A: Memory Allocation for Tasking Environments

A.0 Allocation Algorithm

The algorithm used to allocate activation records is an important feature in any approach used to the management of environments in the presence of tasking. The algorithm must give performance reasonably close to that of a simple stack algorithm for the important subclass of Ada programs which do not employ tasking. At the same time, it must efficiently handle multitasking and the allocation of activation records whose lifetimes are not always conveniently nested.

Dynamic allocation of activation records on each procedure call is an obvious solution to the problem of non-nested lifetimes. However, it is an inherently expensive solution, and imposes a significant execution penalty on programs that do not require its generality. Even with multitasking, dynamic allocation is more general than actually required, since procedure activations remain "last in, first out" within any given task. A better policy is to use stack allocation within any given task as long as possible, with provision to detect overflow from the current region of memory and switch to new region when necessary. This is similar to the approach described, for instance, in [6].

A.1 Simple Frame Management

Figures A-1 and A-3 are Ada packages describing a "stack oriented" allocation algorithm suitable for tasking applications. Figure A-1 describes the "cactus stack" frame management algorithm proper. Figure A-3 describes an associated dynamic memory management algorithm which is invoked by the frame management routines when departures from simple stack management are required. The packages presented are intended only to illustrate the algorithms; they would not be visible in an actual Ada program. They would be implemented in assembly language and used by compiler code generator in subprogram prologue and epilogue code sequences.

In essence, the frame management algorithm works as follows:

1. When execution switches to a new task, a large block of memory is obtained from the memory management system. A minimum size is requested, but no maximum is set; the memory manager simply returns whatever is currently available, so long as it is larger than the specified minimum.

2. The block returned by the memory manager is used as a stack by the task and its procedures until either (a) the task terminates, (b) the stack overflows, or (c)

```
package FRAME_MANAGER is
    procedure GET_FRAME (SIZE: in INTEGER);
    procedure POP_FRAME ();
end FRAME_MANAGER;

package body FRAME_MANAGER is
    with UNCHECKED_CONVERSION;
    with MEMORY_MANAGER;          -- GET_BLOCK, CHANGE_BLOCK, etc.

    with LOW_LEVEL_DEFS;          -- type ADDRESS, WORD_SIZE defined

    type FRAME_HDR is
        record
            SIZE: INTEGER;        -- size of associated frame
            LINK: ADDRESS;        -- ptr to caller's frame
        end record;

    FH_SIZE: constant := 2 * WORD_SIZE;

    type FH_PTR is access FRAME_HDR;
    function FH_PTR is new
        UNCHECKED_CONVERSION (ADDRESS, FH_PTR);

    STACKPTR, LIMIT: ADDRESS;     -- define state of current block
    CURRENT_FRAME: ADDRESS        -- used by generated code

    procedure GET_FRAME (SIZE: in INTEGER) is
    begin
        if STACKPTR + SIZE > LIMIT then          -- block overflow
            GET_BLOCK (SIZE + FH_SIZE, LIMIT, STACKPTR);
        end if;
        FH_PTR(STACKPTR).SIZE := SIZE;
        FH_PTR(STACKPTR).LINK := CURRENT_FRAME;
        CURRENT_FRAME := STACKPTR;
        STACKPTR := STACKPTR + SIZE + FH_SIZE;
    end GET_FRAME;

    procedure POP_FRAME () is
    begin
        TOP: ADDRESS;

        TOP := CURRENT_FRAME + FH_PTR(CURRENT_FRAME).SIZE + FH_SIZE;
        if TOP /= STACKPTR then          -- change current block
            CHANGE_BLOCK (LIMIT, TOP);
        end if;
        STACKPTR := CURRENT_FRAME;
        CURRENT_FRAME := FH_PTR(CURRENT_FRAME).LINK;
    end POP_FRAME;

end FRAME_MANAGER;
```

Figure A-1

```
package FRAME_MANAGER is
    with LOW_LEVEL_DEFS;          -- ADDRESS, etc. defined
    procedure GET_FRAME (SIZE: in INTEGER);
    function EXPAND_FR (INC: in INTEGER) return ADDRESS;
    procedure SHRINK_FR (INC: in INTEGER);
    procedure POP_FRAME ();
end FRAME_MANAGER;

package body FRAME_MANAGER is
    with UNCHECKED_CONVERSION;
    with MEMORY_MANAGER;          -- GET_BLOCK, CHANGE_BLOCK, etc

    type FRAME_HDR is
        record
            SIZE: INTEGER;        -- size of associated frame
            LINK: ADDRESS;        -- ptr to caller's frame
            EXTEN: ADDRESS;       -- link to non-contiguous extension
        end record;

    FH_SIZE: constant := 3 * WORD_SIZE;

    type FH_PTR is access FRAME_HDR;
    function FH_PTR is new
        UNCHECKED_CONVERSION (ADDRESS, FH_PTR);

    STACKPTR, LIMIT: ADDRESS;     -- define state of current block
    CURRENT_FRAME: ADDRESS        -- used by generated code
    CURRENT_EXTEN: ADDRESS        -- normally = CURRENT_FRAME


    procedure GET_FRAME (SIZE: in INTEGER) is
    begin
        NEW_TOP: ADDRESS;

        NEW_TOP := STACKPTR + SIZE + FH_SIZE;
        if NEW_TOP > LIMIT then                    -- block overflow
            GET_BLOCK (SIZE, LIMIT, STACKPTR);
        end if;
        FH_PTR(STACKPTR).SIZE := SIZE;
        FH_PTR(STACKPTR).LINK := CURRENT_FRAME;
        FH_PTR(STACKPTR).EXTEN := 0;
        CURRENT_FRAME := STACKPTR;
        CURRENT_EXTEN := STACKPTR;
        STACKPTR := NEW_TOP;
    end GET_FRAME;
```

Figure A-2.1

```
procedure EXPAND_FR (INC: in INTEGER) is
begin
    NEW_TOP: ADDRESS;

    NEW_TOP := STACKPTR + INC;
    if NEW_TOP > LIMIT then          -- block overflow
        GET_BLOCK (INC + FH_SIZE, LIMIT, STACKPTR);
        FH_PTR(STACKPTR).SIZE := INC;
        FH_PTR(STACKPTR).LINK := CURRENT_EXTEN;
        FH_PTR(STACKPTR).EXTEN := 0;
        FH_PTR(CURRENT_EXTEN).EXTEN := STACKPTR;
        CURRENT_EXTEN := STACKPTR;
        STACKPTR := STACKPTR + INC + FH_SIZE;
        return CURRENT_EXTEN + FH_SIZE;
    else                             -- fits in current block
        TOP: ADDRESS;

        TOP := CURRENT_EXTEN + FH_PTR(CURRENT_EXTEN).SIZE + FH_SIZE;
        if TOP /= STACKPTR then      -- reentering empty block
            NEW_TOP := NEW_TOP + FH_SIZE;
            FH_PTR(CURRENT_EXTEN).EXTEN := STACKPTR;
            FH_PTR(STACKPTR).SIZE := INC;
            FH_PTR(STACKPTR).LINK := CURRENT_EXTEN;
            FH_PTR(STACKPTR).EXTEN := 0;
            CURRENT_EXTEN := STACKPTR;
        else
            FH_PTR(CURRENT_EXTEN).SIZE :=
                FH_PTR(CURRENT_EXTEN).SIZE + INC;
        end if;
        STACKPTR := NEW_TOP;
    end if;
end EXPAND_FR;
```

Figure A-2.2

```
    procedure SHRINK_FR (INC: in INTEGER) is
    begin
        TOP: ADDRESS;

        TOP := CURRENT_EXTEN +
                FH_PTR(CURRENT_EXTEN).SIZE + FH_SIZE;
        if TOP /= STACKPTR then              -- change current block
            CHANGE_BLOCK (LIMIT, TOP);
        end if;
        if INC >= FH_PTR(CURRENT_EXTEN).SIZE then
                                             -- vacating whole exten.
            STACKPTR := CURRENT_EXTEN;
            CURRENT_EXTEN := FH_PTR(CURRENT_EXTEN).LINK;
            if INC > FH_PTR(CURRENT_EXTEN).SIZE then
                SHRINK_FR (INC - FH_PTR(CURRENT_EXTEN).SIZE);
            end if;
        else                                 -- reducing size only
            FH_PTR(CURRENT_EXTEN).SIZE :=
                FH_PTR(CURRENT_EXTEN).SIZE - INC;
            STACKPTR := TOP - INC;
        end if;
    end SHRINK_FR;



    procedure POP_FRAME () is
    begin
        TOP: ADDRESS;

        TOP := CURRENT_EXTEN +
                FH_PTR(CURRENT_EXTEN).SIZE + FH_SIZE;
        if TOP /= STACKPTR then              -- change current block
            CHANGE_BLOCK (LIMIT, TOP);
        end if;
        while CURRENT_EXTEN /= CURRENT_FRAME loop
            CURRENT_EXTEN := FH_PTR(CURRENT_EXTEN).LINK;
            TOP := CURRENT_EXTEN +
                    FH_PTR(CURRENT_EXTEN).SIZE + FH_SIZE;
            CHANGE_BLOCK (LIMIT, TOP);
        end loop;
        STACKPTR := CURRENT_FRAME;
        CURRENT_FRAME := FRAME_PTR(CURRENT_FRAME).LINK;
        CURRENT_EXTEN := CURRENT_FRAME;
        while FH_PTR(CURRENT_EXTEN).EXTEN /= 0 loop
            CURRENT_EXTEN := FH_PTR(CURRENT_FRAME).EXTEN;
        end loop;
    end POP_FRAME;

end FRAME_MANAGER;
```

Figure A-2.3

execution switches to another task. In case (a), the entire block is returned to the memory manager. In case (b), a new block is requested from the memory manager, and the frame for the call causing the overflow is allocated from the new block. The new block is referred to as an overflow block. In case (c), the portion of the block not in use by the current task is returned to the memory manager prior to switching execution.

3. During execution of a task, a procedure return may cause an overflow block to be vacated. For efficiency, the overflow block is not immediately returned to the memory manager. This avoids the need to reallocate the block in case a new procedure call precedes the next return. The global stacktop pointer is left at the base of the vacated block; on a subsequent procedure exit, the inequality of the stacktop pointer with the sum of the current frame pointer and current frame size signals that the frame being released resides outside the overflow block. This means that the overflow block should be released to the memory manager and the block containing the current frame be made the new working block.

A.2 *Frame Management with Extensions*

The frame management algorithm of figure A-1 is overly simple, in that it does not provide for the handling of frame extensions. These are useful in practice to accommodate variable sized structures which cannot be allocated in the basic frame. This is considerably more efficient than the alternative of allocating such structures dynamically from the general purpose data heap. Frame extensions can also be used for temporary storage that is not needed throughout the lifetime of an activation record, increasing storage efficiency.

A frame management routine that provides for frame extensions is illustrated in figure A-2. In addition to the entries GET_FRAME and POP_FRAME of figure A-1, it provides the entries EXPAND_FR and SHRINK_FR to respectively request a frame extension and to release a specified amount of memory from the top of the frame.

The frame extensions handled by the routine in figure A-2 are of two forms--contiguous and non-contiguous. A contiguous extension is simply an additional block of storage contiguous with an existing frame or frame extension. The effect of allocating a contiguous extension is simply to increase the value in the size field of the associated frame or frame extension.

When there is insufficient room in the current working block to allocate a contiguous extension of a requested size, a non-contiguous extension must be allocated. An overflow block is

created, the requested extension is allocated from that block, and a pointer to the extension is stored in the extension link field of the preceding frame or frame extension. The format of a non-contiguous extension is identical to that of a regular frame, except that its link field is a back pointer to the frame or frame extension to which it is attached, rather than to the caller's frame. On a return from a procedure, all non-contiguous frame extensions are released prior to releasing the basic frame of the exiting procedure.

A.3 Dynamic Memory Management

A memory management policy that would appear to work well in support of the routines of figures A-1 and A-2 is a derivative of Donald Knuth's "boundary tag method" [12]. An Ada package implementing the policy is shown in figure A-3. The data structure on which it is based is shown in figure A-4. The implementation assumes byte addressing with four-byte words. Allocations are in integral words, so the two low order bits of the block size specifier in the block control word are available. One of these records the busy/free status of the block above the control word, and the other the block below. Although the basic boundary tag method assumes block size specifiers both above and below each block, the upper block size specifier is needed only if the block is free. It can therefore be allocated as the last word of the block itself, limiting memory overhead for control words to one word per block.

An important feature of the memory management algorithm illustrated is the method in which the block free list is maintained. At the start of program execution, the free list will presumably consist of a single block representing all the memory reserved for allocation of activation records. As task switches cause portions of this block to be broken off, the space begins to fragment. However, the free list is maintained as a circular list with a rotating head. Released blocks are initially inserted at the head of this list, but when a block becomes too small to satisfy a request, the list head pointer advances over it, leaving it at the tail of the list. The result is that dwell time for small blocks prior to reexamination is maximized. This means that there is maximum opportunity for merges with adjacent blocks before the head pointer returns to the block, so that free blocks near the head of the list to be as large as possible.

It is important to the efficient operation of this algorithm in support of frame management that memory contained on its free list be used only for allocation of activation records. If the same free list is used to satisfy requests for dynamically allocated program structures, fragmentation becomes much more serious. It is also helpful if a reasonable minimum ratio between free space and allocated space is maintained. If the ratio falls below about 33%, the memory manager should probably request addi-

```
package MEMORY_MANAGER is
    with LOW_LEVEL_DEFS;

    procedure GET_BLOCK (
        MIN_SIZE: in INTEGER;
        BLOCKTOP: in out ADDRESS;
        STACKTOP: in out ADDRESS;
    );  --  handles oferflow from current block

    procedure CHANGE_BLOCK (
        BLOCKTOP: in out ADDRESS;
        STACKTOP: in ADDRESS
    );  --  for procedure return causing change in current block

    procedure CLOSE_TASK_STORE (
        BLOCKTOP: in ADDRESS;
        STACKTOP: in ADDRESS;
        FLH_SAVE: out FBC_PTR;
    );  --  prepare storage for suspension of task

    procedure OPEN_TASK_STORE (
        BLOCKTOP: out ADDRESS;
        STACKTOP: in ADDRESS;
        FLH_SAVE: in FBC_PTR;
    );  --  prepare storage for execution of suspended task

end MEMORY_MANAGER;

package BLOCK_CTL is
    type CTL_WORD is limited private;
    function BL_SIZE (C: CTL_WORD) return INTEGER;
    function ABOVE_FREE (C: CTL_WORD) return BOOLEAN;
    function BELOW_FREE (C: CTL_WORD) return BOOLEAN;
    procedure SET_BLSIZE (C: CTL_WORD; S: INTEGER);
    procedure SET_ABOVE_FREE (C: CTL_WORD);
    procedure SET_BELOW_FREE (C: CTL_WORD);
    procedure SET_ABOVE_BUSY (C: CTL_WORD);
    procedure SET_BELOW_BUSY (C: CTL_WORD);
private
    type CTL_WORD is new INTEGER;
end BLOCK_CTL;
```

Figure A-3.1

```
package body MEMORY_MANAGER is
    with UNCHECKED_CONVERSION;
    with BLOCK_CTL;

    type FREE_BLOCK_CTL_REC;
    type FBC_PTR is access FREE_BLOCK_CTL_REC;
    type FREE_BLOCK_CTL_REC is
        record
            PREV: FBC_PTR;          -- may be overstored in busy block

            NEXT: FBC_PTR;          -- may be overstored by unused byte
                                    -- count on closing current block;
                                    -- otherwise preserved as free list
                                    -- ptr; block limit ptr points here

            CTL: CTL_WORD;          -- size of block below, plus busy/
                                    -- free bits for blocks above and
                                    -- below; only word always valid

            ABOVE_SIZE: INTEGER;            -- overstored with data if
                                            -- associated block is busy

        end record;


    procedure GET_BLOCK (
        MIN_SIZE: in INTEGER;
        BLOCKTOP: in out ADDRESS;
        STACKTOP: in out ADDRESS;
    ) is
    begin
        PTR1, PTR2: FBC_PTR;
        UNUSED_BYTE_COUNT: INTEGER;
        --
        --  get free list head ptr from current block and store
        --  unused byte count at top of stack
        --
        PTR1 := BLOCKTOP - 1 * WORD_SIZE;
        PTR2 := PTR1.NEXT;                  -- (free list head ptr)
        if PTR2 = NULL then                 -- free list empty
            raise MEMORY_ALLOC_EXCEPT;
        end if;
        PTR1 := PTR2;
        UNUSED_BYTE_COUNT := BLOCKTOP - STACKTOP;
        INT_PTR(STACKTOP).ALL = UNUSED_BYTE_COUNT;
```

Figure A-3.2

```
    --
    --  find new block of suitable size; allocate for stack use
    --
    while BL_SIZE (PTR2.CTL) < MIN_SIZE loop
        PTR2 := PTR2.NEXT;
        if PTR2 = PTR1 raise MEMORY_ALLOC_EXCEPT;
    end loop;
    if PTR2.NEXT = PTR2 then          -- taking last free block
        PTR2.NEXT := NULL;
    else
        PTR2.PREV.NEXT := PTR2.NEXT;          -- unlink from free list
        PTR2.NEXT.PREV := PTR2.PREV;
    end if;
    SET_BELOW_BUSY (PTR2.CTL);        -- mark block busy
    PTR1 = PTR2 - BL_SIZE (PTR2.CTL);
    SET_ABOVE_BUSY (PTR1.CTL);
    BLOCKTOP := PTR2 + 1 * WORD_SIZE;
    STACKTOP := BLOCKTOP - BL_SIZE (PTR2.CTL);
end;


procedure CHANGE_BLOCK (
    BLOCKTOP: in out ADDRESS;         -- input value is old block
    STACKTOP: in ADDRESS;             -- points into new block
) is
begin
    PTR1, PTR2: FBC_PTR;
    --
    --  set free list head ptr in new block to block released
    --  (inserts old block at head of free list)
    --
    PTR1 := FBC_PTR(STACKTOP + UNUSED_BYTE_COUNT - WORD_SIZE);
    PTR1.NEXT := FBC_PTR(BLOCKTOP - WORD_SIZE);
    --
    --  set return parameter to top of new block
    --
    BLOCKTOP := PTR1 + WORD_SIZE;
    --
    --  free old block
    --
    PTR1 := PTR1.NEXT;
    PTR1.PREV := PTR1.NEXT.PREV;      -- (free list tail ptr)
    SET_BELOW_FREE (PTR1.CTL);        -- mark freed block free
    PTR2 := PTR1 - BL_SIZE(PTR1.CTL);
```

Figure A-3.3

```
    if BELOW_FREE (PTR2.CTL) then    -- merge freed block with
                                     -- one below, prev. freed
        NEW_SIZE: INTEGER;

        NEW_SIZE := BL_SIZE (PTR1.CTL) + BL_SIZE (PTR2.CTL);
        SET_BLSIZE (PTR1.CTL, NEW_SIZE);
        PTR2 := PTR1 - NEW_SIZE;
        SET_BLSIZE (PTR2.CTL, NEW_SIZE);
    else
        SET_ABOVE_FREE (PTR1.CTL);
    end if;
end CHANGE_BLOCK;


procedure CLOSE_TASK_STORE (
    BLOCKTOP: in ADDRESS;
    STACKTOP: in ADDRESS;
    FLH_SAVE: out FBC_PTR;
) is
begin
    OLD_SIZE, NEW_SIZE, RES_SIZE: INTEGER;
        -- old, new, and residual block sizes
    UNUSED_BYTE_COUNT: INTEGER;
    PTR1, PTR2: FBC_PTR;

    UNUSED_BYTE_COUNT := BLOCKTOP - STACKTOP;
    PTR1 := FBC_PTR (BLOCKTOP - WORD_SIZE);
    RES_SIZE := UNUSED_BYTE_COUNT - WORD_SIZE;
        -- size of free part of current block, if split
    if RES_SIZE >= MIN_SPLIT_SIZE then
        --
        --  split block into one busy, one free block
        --
        OLD_SIZE := BL_SIZE (PTR1.CTL);
        NEW_SIZE := OLD_SIZE - UNUSED_BYTE_COUNT;
        SET_BLSIZE (PTR1.CTL, RES_SIZE);
        SET_BELOW_FREE (PTR1);
        PTR2 := STACKTOP - WORD_SIZE;
        SET_ABOVE_FREE (PTR2.CTL);
        PTR2.ABOVE_SIZE := RES_SIZE;
        SET_BELOW_BUSY (PTR2.CTL);
        SET_BLSIZE (PTR2.CTL, NEW_SIZE);
        INT_PTR (STACKTOP).ALL := 0;        -- unused byte count
        --
        --  insert free portion of block in free list
        --
        PTR1.PREV := PTR1.NEXT.PREV;
        PTR1.NEXT.PREV := PTR1;
        PTR1.PREV.NEXT := PTR1;
```

Figure A-3.4

```
            --
            --  return parameter saves free list while task inactive
            --
            FLH_SAVE := PTR1;
        else
            FLH_SAVE := PTR1.NEXT;
            INT_PTR (STACKTOP).ALL := UNUSED_BYTE_COUNT;
        endif;
    end CLOSE_TASK_STORE;


    procedure OPEN_TASK_STORE (
        STACKTOP: in ADDRESS;
        BLOCKTOP: out ADDRESS;
        FLH_SAVE: in FBC_PTR;
    ) is
    begin
        PTR: FBC_PTR;
        NEW_SIZE: INTEGER;

        PTR := STACKTOP + INT_PTR (STACKTOP).ALL - WORD_SIZE;
        if ABOVE_FREE (PTR.CTL) then
            --
            -- merge with upper blk
            --
            PTR := PTR + PTR.ABOVE_SIZE + WORD_SIZE;
            NEW_SIZE := PTR.ABOVE_SIZE + BL_SIZE (PTR) + WORD_SIZE;
            SET_BLSIZE (PTR.CTL, NEW_SIZE);
            --
            -- unlink from free list
            --
            PTR.PREV.NEXT := PTR.NEXT;
            PTR.NEXT.PREV := PTR.PREV;
            SET_BELOW_BUSY (PTR.CTL);
        end if;
        --
        --  set up new block for stack use
        --
        PTR.NEXT := FLH_SAVE;
        BLOCKTOP := PTR + WORD_SIZE;

    end OPEN_TASK_STORE;

end MEMORY MANAGER;
```

Figure A-3.5

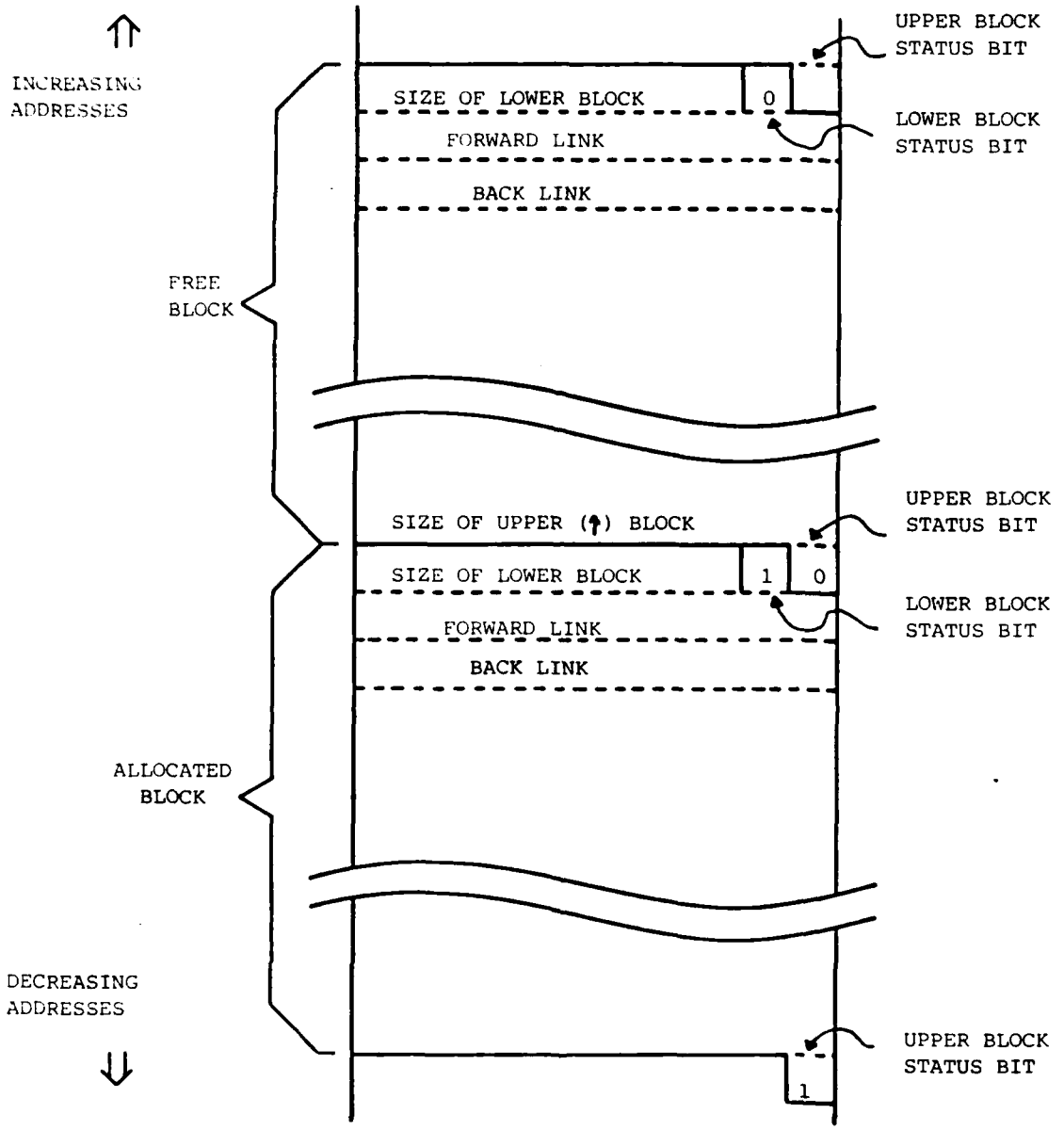


FIGURE A - 4: DYNAMIC STORAGE DATA STRUCTURE

tional system memory. This provision is not included in the algorithm of figure A-3, but would probably be a desirable addition to any actual implementation in support of tasking.

References

1. Reference Manual for the Ada Programming Language, U. S. Department of Defense MIL-STD 1815.

2. Instruction Set Architecture for the Military Computer Family, U. S. Department of Defense MIL-STD 1862, May 28 1980.

3. Report of the MIL-STD 1862 Review Committee, Electronic Industries Association Engineering Department, June 1981.

4. Arnold, R. D., "Ada and the Nebula Architecture", Boeing Aerospace Corporation research note, February 1981.

5. Birtwistle, G. M., Dahl, O.-J., Myhrhaug, B., and Nygaard, K., Simula Begin, Studentletteratur, Sweden; Auerbach Publishers Inc., Philadelphia, Pennsylvania, 1973.

6. Bobrow, D. G., and Wegbreit, B., "A Model and Stack Implementation of Multiple Environments", Communications of the ACM, Vol. 16, No. 10 (October 1973), pp 591-603.

7. Davis, M., and Stryker, D., "Nebula as a Target for Ada", Intermetrics, Inc. report IR #655, February 20, 1981.

8. Haberman, A. N., and Nassi, I. R., "Efficient Implementation of Ada Tasks", Digital Equipment Corporation Research Note

9. Hanson, D. R., "A Simple Variant of the Boundary-tag Algorithm for the Allocation of Co-routine Environments", Information Processing Letters, Vol. 4, No. 4 (January 1976), pp. 109-112.

10. Hirschberg, D. S., "A Class of Dynamic Memory Allocation Algorithms", Communications of the ACM, Vol. 16, No. 10 (October 1973), pp 615-618.

11. Ichbiah, J. D., Barnes, J. G. P., Heliard, J. C., Krieg-Brueckner, B., Roubine, O., and Wichmann, B. A., "Rationale for the Design of the ADA Programming Language", ACM SIGPLAN Notices, Vol. 14, No. 6 (June 1979) Part B.

12. Knuth, D. E., The Art of Computer Programming; Volume 1: Fundamental Algorithms, Addison-Wesley Pub. Co., Reading, Massachusetts, 1973.

13. Marlin, C. D., Co-routines: A Pragramming Methodology, a Language Design, and an Implementation, Lecture Notes in Computer Science, Vol. 95, Springer-Verlag, 1980.

14. Marlin, C. D., "A Heap-based Implementation of the Programming Language Pascal", Software Practice and Experience, Vol. 9, No. 2 (February 1979), pp. 101-119.

15. McMinn, C., Markowitz, R., Wharton, J., and Grundmann, W., "Silicon Operating System Standardizes Software", IEEE Electronics, September 8, 1981, pp 135-139.

16. Wolf, W., "Compilers and Computer Architectures", IEEE Computer, Vol. 14, No. 7 (July 1981) pp 41-47.

Writing Portable Programs

for the Nebula ISA

Steven L. Worona

Digicomp Research Corp.

Ithaca, NY

References to Nebula in this report refer to the version of MIL-STD-1862A dated "TBD"

Implementing Ada on the Nebula Architecture:

Design Issues and Alternatives

R. D. Arnold

Boeing Aerospace Corp.

February 1982

References to Nebula in this report refer to the version of
MIL-STD-1862A dated 1 July 1981

PART 1 - BACKGROUND

The Nebula Instruction Set Architecture (ISA) is a 32-bit, general-register design originally developed for use in military embedded-computer applications. Nebula began as part of an effort "to provide the US Army with a family of instruction-set-compatible computers of varying performance capabilities" [1]. Initially, this "Military Computer Family" (MCF) project attempted to select an existing computer architecture as the Standard. After a 2-year analysis of a number of commercial and military machines, the Army selected in 1973 a "'best' commercial architecture to fit its tactical requirements" [2]. Proprietary issues, however, eventually led to the abandonment of this approach. In September, 1979, the Army, through CORADCOM, contracted with a group at Carnegie-Mellon University to develop a new architecture, now known as "Nebula". The initial Nebula specification, published as MIL-STD-1862 on May 28, 1980, has since been revised several times; the version upon which this report is based is MIL-STD-1862A, "Date TBD" [3], issued in late September, 1981.

Air Force involvement with the Nebula project dates from September, 1980. Although both services share the goal of developing a hardware Standard for embedded systems, the planned Air Force uses and acquisition policies vary significantly from those of the Army. In considering questions of software portability, it is essential to note how the Air Force and Army differ in their plans for multi-vendor participation:

- The Army will procure up to a 5-year supply of Nebula computers from a single source. Design efforts are now under way by four computer manufacturers -- IBM, RCA, Raytheon, and (jointly) GE

and TRW -- to construct "Advanced Development" machines meeting the Nebula specifications. The Army will evaluate these machines for performance, cost, etc., and select two of the vendors to build "Engineering Development" models. After a final analysis by the Army, one of these two machines will be chosen, and competitive bids (open to all vendors, not just those previously involved) solicited for up to a 5-year supply of that machine. At that point, the ISA may be reviewed and modified, and another round of competitive design and implementation will begin. (The first production units based on the initial Nebula design are scheduled to be fielded in 1987.) By such periodic reviews and re-designs, the Army hopes to have incorporated within its embedded systems state-of-the-art technology while also maintaining a standard ISA.

- The Air Force, on the other hand, intends to have continuous competition among vendors for ever faster, cheaper, and more reliable computers that implement a standard ISA. An "accreditation approach" is planned, through which any supplier's machine may be acquired so long as it satisfies the specifications of the Standard.

Because of this difference in approach, the Air Force must place very heavy emphasis on the clarity, precision, and completeness of its ISA Standard, while the Army may be prepared to demand far less in these areas. This paper is part of an effort commissioned by the Air Force to help determine whether Nebula meets its requirements for an ISA Standard.

PART 2 - CONCEPTS OF PORTABILITY

2.1. Nebula's Goals

The "Scope" and "Purpose" sections of MIL-STD-1862A read, in their entirety:

> This standard defines the Nebula Instruction Set Architecture. The instruction set architecture includes all information required by a programmer in order to write any time independent program that will execute on computers conforming to this standard.
>
> The purpose of this document is to define the Nebula Architecture, independent of any specific implementation or vendor, with sufficient precision to permit independent implementations of this architecture that execute identical programs in the identical manner. [3, p.1]

That is, a fundamental goal of MIL-STD-1862A is to define an ISA that supports program portability: The ability of a given program to be moved from machine to machine and function identically -- and correctly -- on all of them. The primary concern of this paper is the degree to which Nebula meets this objective.

In order to allow vendors maximum flexibility in the implementation techniques and technologies used for Nebula, MIL-STD-1862A specifies certain details of the ISA as being variable from machine to machine. To avoid conflict with the portability goal described above, the Nebula designers sought to "reduce the visibility of the hardware to the software" [1, p.35]. That is, inter-machine differences would be unimportant -- they would not affect program portability -- since they would not be software-visible. An implicit concern of this paper, then, is the extent to which the hardware variability permitted by MIL-STD-1862A is truly software invisible.

It is worth noting that the importance of software visibility ("These

visibility goals underlie much of the structure of the architecture adopted for the military computer family" [1, p.35]) can be traced back to the original Army effort to select an existing architecture for the MCF:

> With a well specified architecture, details of data bus width, technology (core memory versus semiconductor memory, TTL versus ECL circuits), implementation speedup techniques such as cache memories and instruction lookahead buffers, physical size of computer, etc. need not be of concern to the programmer. A clear (and clean) distinction between the architecture and implementation detail allows software to be transported between computers with the same architecture even though they may have very different implementations. [4, p.4]

2.2. Excluded Issues

Leaving Nebula aside for the moment, there are certain universally accepted cases in which any two computers -- C1 and C2 -- with the same ISAs may fail to run identical programs identically:

- Type and Location of I/O Devices. If, for example, C1 has a tape drive at device address X and C2 has a card reader there -- or even no device at all -- a large class of programs will operate differently on the two machines.
- Physical Memory. If C1 and C2 have different amounts of memory, or if the attachment of the memory modules leaves "holes" of undefined addresses at different points in the address spaces of the two machines, then a program with access to real memory may not run the same on each.
- Operating System. If C1 and C2 are running different operating systems when the compatibility test is made, almost any program will run differently -- if it runs at all -- on the two machines.

- Timing. There are a great many ways that multiprogramming load, I/O-interrupt frequency, time-of-day-clock setting and the like can influence the operation of a program. (Certain programs -- e.g., operating systems -- are intentionally constructed to recognize and be influenced by such phenomena.) These effects may be observed even between multiple executions of a given program on a single system; they may be even more pronounced when the executions take place on physically distinct systems.
- CPU ID. Certain machine architectures provide for a software-accessible identification code that uniquely designates each individual CPU. A program written to examine this data can be made to perform differently depending upon the value found.

Inter-machine differences of the types described above are not the subject of this investigation. In all of the discussions that follow, we will assume that such differences do not exist.

2.3. Categories of Machine Dependencies

Given two machines conforming to the Nebula ISA Standard, the differences between them may be divided into two categories:

1. Those documented as permissible within MIL-STD-1862A.
2. Those resulting from contradictions, ambiguities, or omissions in MIL-STD-1862A.

If any other differences exist, then one of the machines must violate the Standard in some way.

Differences in Category 1 are described in MIL-STD-1862A by several terms, including "implementation dependent", "undefined", and "unpredictatable". The word "reserved" is also used to indicate areas

in which the Standard allows inter-machine differences. In certain other cases, MIL-STD-1862A simply lists acceptable alternatives, as in: "...a Segment.Specifier exception or a hard memory error trap shall be initiated, depending upon the implementation" [3, p.145].

Cases in Category 2 are far more numerous than those in Category 1, arising primarily in the areas of exception and trap handling, the task and procedure interfaces, and the memory management system. Nebula fails to specify, for example, the rules for virtual-address computation, the values of result operands when certain exceptions are raised, the effect of overlapping operands (except in a few cases), the interaction of memory management traps with operand addressing errors and of instruction-execution exceptions with both of these, etc. Each instance of unspecified machine behavior is an implicit implementation dependency: No matter what a machine does in such a case, it may still be certified as adhering to the "Standard" ISA.

The apparent reason for Nebula's silence in these and other areas is the desire to avoid overly constraining the implementors. Whether or not this is legitimate justification for the number of machine dependencies in MIL-STD-1862A, these machine dependencies need to be made explicit. That is, all inter-machine differences allowed by MIL-STD-1862A should be in Category 1; Category 2 should not exist.

The architecture of the IBM System/360 was designed to provide the sort of implementation independence across a wide range of machines sought by Nebula. IBM has produced over 40 different models based on this and the extended System/370 architectures, with a performance ratio of up to 450:1 between the most and least powerful. In addition, other vendors have successfully designed and manufactured mainframes that are plug-compatible with these IBM systems. It is therefore worth noting

the following comments contained in a recent article by one of the System/360 designers:

> In order to ensure compatible implementations, the architecture has to be complete in that it must cover all functions of the machine that are observable by the program, including all the unlikely concurrent occurrences of different unusual exceptions. It either must specify the action the machine performs or state that the action is unpredictable...
>
> The System/360 architecture did not provide adequate precision and detail in some areas. Because there was no specification of the priority in which concurrently existing program exceptions are recognized, programming of virtual machines was made difficult. Because the sequence and concurrency for storage accesses were not specified, processors could not communicate reliably using shared main storage. And because not enough details in machine-check handling were specified, the possibility of model-independent recovery after an equipment failure was reduced. [5, p.384]

Not all instances of ambiguity or self-contradiction in MIL-STD-1862A have a direct bearing upon software portability. It is important to note that these exist, however, since they contribute to the overall level of imprecision of the document. Consider Section 5, "Operand Addressing Modes", in which Nebula's 12 operand specifiers are defined and described. The operand specifiers

> contain the information to determine the location and the size of the operands to be accessed... The location (address) of all operands must be computable in the absence of any context information provided by the opcode. This permits operands to be "pre-evaluated" in the absence of such information... [3, p.7]

While this is certainly a valuable property for an ISA to have (see [6]), there are at least 32 Nebula instructions with operand formats that are "special cases": not represented by an operand specifier. No operand pre-evaluation is possible without at least enough context information to know that the opcode is not one of these 32. Furthermore, for 3 Nebula instructions, determining the format of

operands after the first one requires examining a separate value in memory, the address of which is specified by the first operand.

2.4. Detecting Machine Dependencies in Nebula Programs

As already mentioned, it is quite possible -- in fact, trivially easy -- to write Nebula programs that fail to operate identically on two different machines conforming to MIL-STD-1862A. On the other hand, it is also possible to write a large number of programs that may be moved from machine to machine without detectable change. The important issue here is how to tell which of these categories a particular Nebula program belongs to.

For almost all user programs, the execution environment is established only partially by the machine architecture; a significant component is provided by the operating system. Given that it is possible to write non-portable Nebula programs, it is natural to ask whether a Nebula operating system could be built to limit user programs to only the machine-independent features of Nebula. If so, the problems caused by implementation dependencies in MIL-STD-1862A might be minimized, since these would show up as "bugs" during software testing.

The Nebula designers clearly had such a software-enhanced execution environment in mind, as indicated by the following statements:

> This restriction [on the effects of Nebula actions characterized as "unpredictable"] may be breached by a supervisor that gives a user access rights to critical memory or functions. For this reason, access to the CPU registers in the I/O space should be controlled by means of the memory management system. [3, p.3]
>
> The memory management system provides a mechanism for protecting against such invalid software actions [as reading or writing parts of the context stack]. [3, p.25]

> In time critical applications that are deeply nested, an exception handler should be inserted every few levels to insure proper response to interrupts. [3, p.36]

Although an operating system might well be able to deal with these specific situations -- i.e., denying the user access to the context stacks and I/O space, and ensuring availability of an interrupt handler -- the vast majority of machine dependencies cannot be masked from user programs by the operating system. (See especially Section 3.1, "Some Fundamental Questions", below.) Moreover, even if such an operating system could be constructed, it would not itself be implementation independent: Each Nebula implementation would require its own version of an operating system designed to mask its implementation dependencies -- an example of the sort of duplicate effort which should be eliminated by a standard ISA. The cost of such an operating system -- both initial development and subsequent maintenance -- should properly be included in the life-cycle cost of the hardware.

An operating system does not represent the only technique available for hiding machine dependencies. An assembler, for example, might be designed to prohibit -- or at least flag -- instructions whose execution could vary among Nebula implementations. Compilers might be constrained to generate only code sequences whose behavior was consistent across the entire range of Nebula machines. Again, however, the imprecision of MIL-STD-1862A makes the possibility of constructing such assemblers and compilers doubtful. And it would certanly be impossible to verify that a compiler or assembler did, in fact, make Nebula's machine dependencies software invisible. Similarly, there can be no set of programmer guidelines whose observance would guarantee software portability across Nebula machines.

2.5. The Question of Errors

Many of the inter-machine dependencies cited below have to do with programmer errors, and it might be said that programmers should simply avoid making mistakes whose consequences are machine-dependent (or accept machine dependencies as one of the consequences of an error). Such a response is unacceptable, however, for the following reasons:

- However hard a programmer tries, errors are inevitable. If a "Standard ISA" allows non-standard error responses, the programmer must be familiar with each machine's particular idiosyncrasies before fully understanding how to debug his programs. With debugging representing a significant part of the cost of software development, a Standard ISA cannot afford to leave error responses to each vendor's taste.
- It is one thing for machines to provide different forms of notification when faced with a particular error case; with sufficient study or perseverence, the programmer will eventually uncover the source of the failure and correct it. But what if the mistake is such that one "Standard" machine recognizes it and one does not? And what if development is carried out on the more tolerant machine, while the target computer, the one in the aircraft, say, is the one that traps on the error? Imagine, for example, that the programmer develops a subroutine on a Nebula which -- in full conformance with MIL-STD-1862A -- is generously silent about the insertion of non-0 bits in some or all of the architecture's "reserved" bit fields. It is unlikely that such a bug would be uncovered before the software is installed (and

fails) in the "Standard" embedded system in the field.

In light of these considerations, the following passage from the definition of "Unpredictable" in MIL-STD-1862A should be noted:

> An unpredictable action may produce any change in the state of the machine that is consistent with the rights of the program that caused it. For example, an unpredictable operation performed by a user task may destroy any of the locations it can access normally, but shall not destroy any state protected by the protection mechanisms... It should be noted that while the programmer cannot rely on any properties of an unpredictable action, it is considered desirable to make such actions as innocuous a practical. [3, p.3]

The final sentence is the opposite of what is required for portability: The more innocuous an "unpredictable" action is on a given Nebula computer, the more likely that such an action will be incorporated, undetected, within a Nebula program.

For the purposes of software portability, Nebula should be designed so that

1. A particular operation will raise an error indication in all machines or none, and
2. The same error indication will be raised in all machines.

2.6. Two Examples

As one of the industry's most widely copied architectures, it is not surprising that the IBM System/360 design has generated a number of examples relevant to the discussion of software portability. In both of the cases described below, "plug-compatible" machines turned out not to be.

1. The BXH instruction ("Branch on Index High") is usually used on IBM System/360 and System/370 machines to control loops. It

includes three register operands: a base, an increment, and a comparand. The base is increased by the increment, and a branch is taken depending upon the relation of the sum to the comparand. If the base and comparand registers coincide, the system architecture descriptions [7, 9] state that "the original contents are used as the comparand". Thus, by specifying the same register as base, increment, and comparand, it is possible to use a single instruction to double a register and branch if its original contents were positive. At least one model, however, of an IBM-plug-compatible series of machines manufactured by ICL-Dataskil (London) in the late 1960's failed in this case to compare the updated register with its original contents. System/360-targetted software depending on this feature -- and there was some -- did not operate properly on these machines.

2. The TRT ("Translate and Test") instruction in IBM's architecture is similar to Nebula's SCANB: Each byte in a source string is used as an index into a table (a bit table in Nebula, a byte table in System/360-370), and the table value determines whether the next source byte is to be processed. The TRT instruction includes a length specification for the source string, which gives an upper bound on the number of bytes to be processed. (Of course, the actual number of bytes processed depends upon the contents of the string and table.) It is common programming practice to append a "sentinel byte" -- one whose table value will stop the scan if no prior byte does -- to the source string. The maximal string length (256) may then be used, and no length calculations are required.

This technique works, in part, because the IBM System/370 architecture specifies that only those source-string bytes preceding a "break byte" are actually fetched from memory. No memory-reference exceptions, therefore, may be caused by the portion of the source string beyond the sentinel. A "plug-compatible" machine manufactured by Siemens AG (Munich), however, tested to see that the pages containing the entire source string, as defined by the length in the TRT instruction, were accessible before beginning execution of the instruction. In the rare cases when a source string was located close to the end of the virtual address space of a process, an unanticipated page fault might occur.

These anecdotes help emphasize the need for careful analysis of the operation of any machine that is supposed to function according to a particular architectural specification. However, no such analysis is possible unless the specification itself is clear, precise, and complete.

In addition, the examples exhibit two different levels of detectability for machine dependencies. Once the case of the BXH incompatibility became known, assemblers for ICL machines could easily have been modified to detect and flag any instances of the problem instruction. With the TRT problem, however, there is no way for a programmer to automatically be kept from intentionally or inadvertently writing machine-dependent code. The TRT anomaly, therefore, is far more dangerous, and would be far more important to correct. It must be noted that the great majority of inter-machine differences allowed by MIL-STD-1862A are of the latter variety.

2.7. "Controlled Non-Portability"

Part 3 of this paper details many of the ways that a given program may function differently on two different Nebula computers. Along this vein, it will be shown that MIL-STD-1862A explicitly and implicitly allows implementors too much flexibility for the software portability that was one of its primary goals.

Although the purpose of this study was to identify all portability-related Nebula problems, it is not the case that an acceptable ISA Standard must allow absolutely no inter-machine differences. A designer of the IBM System/360 architecture has observed:

> Identical action in all machines is less likely to cause problems with compatibility and has a certain aesthetic appeal. Indiscriminately specifying predictable operation, however, may present problems when the predictable operation is of insignificant value to the user and some later machine has difficulty complying with the required predictability. Whereas specifying initially that an operation is unpredictable might have been quite acceptable, relaxing the architecture definition to permit unpredictability has certain risks, because some programs may have come to depend on the initial, precise definition. Thus the architect has to make a deliberate decision about the extent of predictability. [5, p.384]

The key phrase here is "deliberate decision", leading to what might be called "controlled non-portability".

In order for deliberate decisions to be made about potential implementation dependencies, it is first necessary to have a very precise definition of the ISA which is not implementation dependent. There must be no ambiguities in such a definition, no obscure cases whose results are left unspecified. Only then can a careful, rational, deliberate decision be made regarding any proposals to relax a particular requirement so as to allow inter-machine variation.

Useful consideration of a proposal to allow a machine dependency requires an analysis of its costs and benefits. How likely is it that a programmer will stumble into the resulting portability problem? Is there any way of automatically detecting the problem at compilation or assembly time? Can an operating system rule out the problem during execution? What specific payoff -- e.g., cheaper (how much cheaper?) or faster (how much faster?) hardware -- will actually result?

Starting from a suitably complete ISA Standard, the Nebula Review Board (or some technical group appointed by the Board) might perform such an analysis for any vendor's proposal to relax the Standard. In this way, only those implementation dependencies with sufficiently low cost/benefit ratios would ever exist, and those that did exist could be carefully documented and perhaps even monitored by operating systems, compilers, and assemblers.

The idea of strictly controlling architectures and implementations so as to limit software portability problems is not a radical or new one; its use in commercial architectures provides reason to expect a high payoff. Again considering the IBM System/360:

> A set of procedures have been established for the development of an architecture, starting with the conception of the idea and ending with the formal adoption of a definition. These procedures provide for the assessment of the cost and value of a function and for the approval of the architecture by machine and software implementers. Rules have been established about the extent of architectural compatibility, and provision is made for deviating from the common definition.
>
> Although the implementation of a line of compatible computers did not take an undue amount of effort, the design and control of architecture proved to require more attention to detail than originally anticipated. Furthermore, experience with System/360 and its subsequent extensions has shown that the management of architecture must be an ongoing operation to ensure that the evolution of the architecture structure is governed by a consistent set of principles and a design philosophy. [5, p.385-386]

Similar safeguards have been taken for the architecture of the VAX family of computers:

> The goal of achieving software compatibility in a family of machines requires a certain amount of discipline in the design process that shapes the underlying architecture... Because software compatibility over a wide range of implementations was a major design goal for Digital Equipment Corporation's VAX series of computers, architecture management was put in place during the early stages of design... The emphasis is clearly on maintaining architectural stability by eliminating gratuitous differences... Future enhancements will be incorporated into the architecture in a carefully controlled manner... The architecture management function is delegated to a centralized organization that is not biased by the parochial needs of specific hardware or software development projects. Architecture management activities center around both the active maintenance of the architecture control document and a well-defined process for the consideration of changes to its content... The architecture document is intended to be complete and self-sufficient. Whenever ambiguities, contradictions, or missing links are found in the specification, the document is updated. [8, p.87-90]

The relevance of DEC's and IBM's experiences in this area was evident to those who conducted the 1976-1978 evaluation of commercial ISA's as candidates for the Army's MCF:

> By careful family planning, IBM has attempted to insure the transportability of user programs between machine models. While not 100% successful in this goal, IBM has over 15 years of experience and has come closer to the transportability goal than any other manufacturer.
>
> DEC has built a family of PDP-11's, but there are now incompatibilities in the instruction sets. Certain instructions execute differently on different machines. If there is no canonical PDP-11, then all the software may not be transportable throughout the MCF. Family consistency requires careful planning. (IBM has a full-time staff of 12 professionals whose sole job is to insure family consistency). [13, p.10-11]

In summary, then, a standard military ISA should be developed, defined, revised, and monitored at least as carefully and precisely as the best commercial techniques allow.

PART 3 - PORTABILITY PROBLEMS IN NEBULA

This section documents some of the most important, most obvious, and most interesting ways in which a program may fail to execute identically on various Nebula machines. Not all possibilities have been covered. In particular, the entire Nebula I/O specification is left to another paper. Even ignoring I/O, however, the scope of implementation dependencies within Nebula is far too complex to itemize completely.

In many cases, the portability problem is posed in the form of a question: What will a Nebula machine do in the following circumstance? These are generally areas where MIL-STD-1862A fails to indicate how a calculation is to be performed, whether a particular set of events is legal, what sort of exception might be raised, etc. Each implementor will encounter these questions (and many others) in the process of designing and building a Nebula machine; with no guidance from the "Standard", it is certain that different vendors will arrive at different answers.

3.1. Some Fundamental Questions

Because MIL-STD-1862A nowhere defines the "basic machine cycle" of the Nebula architecture, nor specifies how operand evaluation and instruction execution relate to each other, certain basic issues are left ambiguous. Some of these will come up again in later sections, but it is useful to summarize them here.

3.1.1. Calculation of Virtual Addresses

MIL-STD-1862A defines the Architectural Virtual Address Space to be "the number of distinct byte virtual addresses that can be generated by the addressing modes of the architecture" [3, p.43], and specifies its size as 2**32. That is, all Nebula virtual addresses are representable as 32-bit quantities. There is no mention in MIL-STD-1862A, however, of what happens when an address calculation yields a value outside this range.

Address calculations occur in (at least) the following contexts:

- Byte- and word-indexed operand addressing modes.
- Scaled- and unscaled-indexed operand addressing modes.
- Instructions that adjust the stack pointer, including PUSH, POP, RSR, and JSR.
- Most of the "Control Instructions" [3, Section 22], which add 8- or 16-bit inline literals to the Program Counter.
- The MOVTR and SCANB instructions, which use 8-bit quantities to index into a table in memory.
- All of the "Bit Field Instructions" [3, Section 27], which modify a storage address ("Base") by "Pos", an arbitrary positive or negative bit count.
- All instances of vectored operations, where the memory location referenced is generally some number of words beyond (or, in certain cases, one word before) a base address. These include the instructions REPENT, MAP, SETSEG, and SVC, along with all traps and interrupts.

There are several reasonable ways that a vendor might implement virtual-address calculations:

- Perform all virtual-address arithmetic modulo 2**32; that is, keep only the last 32 bits of all calculations.
- Recognize an error whenever a virtual address exceeds 2**32-1 or is negative.
- Treat all virtual addresses that are 2**32 or higher as 2**32-1 and/or all virtual addresses that are negative as 0.

The only reference to this problem in MIL-STD-1862A involves the multiplication necessary to do the scaling for the scaled-index operand addressing mode. In this case, "if the index overflows during scaling, the low order 32 bits are added to the base to form the operand address" [3, p.17]. Rather than helping, however, this isolated reference presents a dilemma: If modulo-2**32 arithmetic is used for index scaling, what different rule (or rules) are in effect for other address calculations? If modulo-2**32 arithmetic is the normal mode (as one might suppose), why make a special statement for index scaling?

The Standard should state clearly how virtual address calculation is to be performed.

3.1.2. Operand Evaluation and References

MIL-STD-1862A states: "Addressing modes are also required to be free of side effects. This eliminates any order dependencies in operand evaluation" [3, p.7]. (The "also" refers to Nebula's provisions to allow "operand pre-evaluation", which have been dealt with previously in this paper.) It is clear, however, that operand evaluation may have some very significant side effects: exceptions and traps. When evaluation of one or more operands of an instruction causes an exception or trap, the order of evaluation is visible to the software.

For portability, it is essential that memory-management traps occur predictably from machine to machine. That is, a program that runs correctly on one version of Nebula should not be able to cause a memory-management trap on a different "Standard" machine. Note that a program may be "correct" even if it involves an exception. Ada, for example, considers exceptions an integral part of its language specification, and several features of Nebula's exception-handling mechanism have been influenced by the Ada requirements. In general, exceptions are conditions that may legitimately be handled by an application program without operating-system intervention; traps, on the other hand, must always involve the operating system.

MIL-STD-1862A violates the requirement for predictability of memory-management traps in at least the following ways:

- Since order of operand evaluation is unspecified, an exception raised by one operand might mask a memory-management trap in another.
- For the CMPBK, MOVBK, MOVM, and MOVTR instructions, it is not specified whether a memory reference occurs (or, in fact, whether the remaining operand specifiers are evaluated at all) in cases when "Cnt" is 0. The same applies to the SCANB instruction when "Slen" is 0, and to the SBF, LBFS, and LBF instructions when "Size" is 0.
- For the MOVBK instruction: "If Src=Dest, the implementation is not required to check or perform memory accesses" [3, p.131].
- For the CMPBK instruction: "If the two blocks are not equal, the implementation is not required to check or perform memory accesses beyond the first nonequal items" [3, p.130].
- In the case of the MOVTR and SCANB instructions, must the

implementation determine the accessibility of all the bytes of the table (256 for "Table" in MOVTR, 32 for "Btable" in SCANB), or only the portion indexed by the bytes in "Src"?

- For the SCANB instruction, must the implementation test the accessibility of all "Cnt" bytes of "Src", or only those bytes actually fetched?
- The CMPS instruction compares the values of two of its operands, and assigns to one or the other depending on the result. Must both operands be write-able, or just the one actually selected for modification?
- May an implementation, as part of pipelining or some other speedup technique, suppress evaluation of an operand specifier if the operand value is not needed? For example, if one operand of the MUL instruction can be quickly determined to be 0, is it legitimate to set the result to 0 immediately? If the divisor in a DIV instruction is 0, may an implementation raise the Illegal.Divisor exception without evaluating the dividend?

For full transportability, exceptions as well as memory-management traps must be predictable. The complexity of the Nebula architecture makes it extremely difficult to specify completely the interaction of all exceptions and traps, short of a full description of the logic of operand evaluation and reference. The reluctance on the part of Nebula's designers to include such a description is understandable.

Although it deals with a somewhat simpler architecture than Nebula, the description of "Serialization" for the IBM System/370 [9] suggests an approach:

> All interruptions [which include what would be called "exceptions" and "traps" in Nebula], and the execution of certain instructions, cause serialization of CPU operation.

> Execution of a serialization function consists in completing all conceptually prior storage accesses by this CPU, as observed by channels and other CPUs, before the conceptually following storage accesses occur. [9, p.28]

One of the merits of such an approach is that there need be no serialization-caused delay in machine execution until an exception or trap occurs. On the other hand, depending on the implementation, a great deal of circuitry might be needed to make this serialization possible, perhaps leading to an unacceptably expensive machine. An analysis of the costs and benefits of a serialize-on-exception/trap capability should be part of the evaluation of proposed Nebula implementations. However, no such analysis can take place for Nebula computers, since MIL-STD-1862A is not precise enough to determine the meaning of the phrases "conceptually prior" and "conceptually following".

Similar topics are considered for DEC's VAX architecture [10], upon which Nebula is loosely based, in [12, p.51-52]. It is worth noting the following description of exceptions on the VAX architecture:

> Both hardware- and software-detected exceptions occur synchronously with the execution of a process. That is, they occur as the result of the execution of a specific instruction sequence; if that sequence were repeated, the same exception would occur again. [11, p.387]

Even this level of repeatability is not assured by MIL-STD-1862A.

3.1.3. Overlapping Operands

MIL-STD-1862A considers overlapping operands for the following instructions:

- EDIV, EXCH, and MAP. Each of these instructions has two output operands. If these operands overlap, MIL-STD-1862A specifies

that the result is undefined.

- MOVBK. "...Overlapping source and destination fields do not affect the results" [3, p.131].
- MOVTR. "If Dest overlaps the translation table, the result is undefined" [3, p.132].

For all of the other Nebula instructions, how will a "Standard" machine treat overlapping operands? MIL-STD-1862A appears to leave the matter open; does this mean that the answer is implementation dependent? For program portability, the Standard should specify that overlapping operands may not affect the results of an instruction, except where explicitly mentioned.

3.1.4. The Program Counter

The program counter is one of the most fundamental components of most computer architectures; its contents and functions are generally defined very thoroughly and precisely. In MIL-STD-1862A, however, there is no formal definition of the program counter. Instead, there are isolated references to its value in certain special cases. There seems to be no underlying model linking these special cases, leading to a confused picture of what the Nebula program counter is and what it does.

Nebula's departure from conventional practice appears, once again, to be aimed at providing maximum flexibility for the implementor. To maintain the software invisibility of the resulting implementation dependencies, the Nebula architecture provides no direct way, in general, for the programmer to test or set the value of the program counter. Nonetheless, the ill-defined nature of Nebula's program counter leads to the following problem areas:

- Although the Nebula design attempts to make the context area software-invisible, at best it simply provides the means by which an operating system may choose to make the context area inaccessible to user programs. With respect to any user programs not so restricted, and with respect to the operating system itself, the context area -- and, therefore, the program counter -- is, in fact, software-visible. (See "The Procedure Interface" below.)
- When a supervisor exception handler is invoked, its third parameter is defined as:

> The program counter of the context that invoked the supervisor exception handler, as a register. This program counter contains the address of the instruction to be executed if the supervisor exception handler does a RET... This parameter may be read or written. Writing this parameter is equivalent to altering the caller's program counter by reference. [3, p.38]

This suggests strongly that the Nebula *program counter* points to the beginning of an instruction. A similar impression is conveyed by the symbolic descriptions of many of the Nebula Control Instructions [3, Section 22]. In the following cases, however, the program counter is described in such a way as to make this interpretation less certain:

- For the register-indexed operand addressing modes, "if the PC is used as the Index Register, it points to the displacement when calculating the memory address" [3, p.12]. This suggests that the program counter "moves over" the various parts of the instruction being executed.
- The CASE instruction includes a list of displacements, one of which is selected and added to the program counter. However, "after operand evaluation the PC is pointing at

displacement[0], not the next instruction. Therefore the branch displacements are relative to the address of displacement[0]" [3, p.109].

- When a new procedure is entered, "space shall be allocated on the context stack for each register starting with 0 (the program counter)... The contents of these newly created registers are undefined with the exception of register 1" [3, p.27]. At what point during procedure entry is the program counter defined? If an exception or trap occurs during the relatively complex process of calling a procedure, where does the program counter point?
- If an instruction crosses a segment boundary in such a way that a memory management trap occurs, what is the program counter in the suspended procedure?
- If the program counter references an improper segment due to a Control Instruction -- that is, if after being incremented by a displacement, the program counter points into a segment that may not contain instructions -- what is the program counter at the time of the memory management trap?
- The JSR and RSR instructions may lead to traps or exceptions; what is the state of the program counter in such cases?
- What is the state of the program counter in a procedure suspended in the middle of an "interruptable" instruction? (See Section 3.2 below.)

Many other questions may be asked whose answers will depend upon how a particular vendor chooses to implement the Nebula program counter. With this degree of ambiguity surrounding such a central component of the architecture, program-counter-related portability problems are likely to arise with each new Nebula implementation.

3.2. Interruptable Instructions

The 5 "String Instructions" [3, Section 26] are required by MIL-STD-1862A to be interruptable. This means:

> If an interrupt or trap occurs during the execution of such an instruction at a point where processing has begun but not yet completed, the intermediate state of the instruction is preserved (in the context stack, in an implementation-dependent form). When the interrupt or trap handler returns and the instruction is resumed, the instruction shall be correctly completed, provided that certain operands of the instruction have not been altered by means other than the interrupted string instruction...
>
> If a string instruction is interrupted before completion, the entire contents of its destination region, as well as any condition codes set by the instruction, are undefined unless and until the instruction is resumed and completed. Moreover, if any source or destination region of a string instruction is altered after the instruction processing has begun but has not yet completed because of an interrupt or trap, or is altered because of any other memory writes not performed by the CPU (such as an I/O transfer) after the instruction processing has begun but has not yet completed, then when the instruction is resumed it shall completely and correctly transfer control to the next instruction, but the contents of any destination region are undefined, and any condition codes or ordinary destination operands set by the instruction are undefined. [3, p.129]

Each implementation dependency and undefined state in the above description is an opening for non-transportability. There are, moreover, additional problems here that are much more subtle:

- Among other things, memory-management traps will be used to indicate the need to read non-resident segments into memory from disk. However, the passages from MIL-STD-1862A quoted above state that such input produces undefined memory locations whenever the trap involves a destination region of a string instruction. This effectively rules out demand-paging systems on

Nebula.

- In the case of either an interrupt or a trap, a task switch may be necessary (for example, to process an interrupt in the context of the task that requested the I/O). May the memory map -- or the map pointer -- be changed while a string instruction is suspended? What is meant by modifying the source or destination regions of a suspended instruction in a suspended task?
- To avoid the possibility of a nonterminating succession of segment faults, an operating system would need to "lock" any memory-resident segments of a suspended string instruction. The MAP instruction can be used to determine the segment in which the trapping memory reference resides, since the trap handler receives its address as ?1. How can the segments of the other operands be determined? The trap handler receives the address of the opcode of the problem instruction (as ?2), but determining the memory locations of its operands may require access to the register and parameter descriptors within the context stack, and these are maintained in an implementation dependent format and location. Furthermore, any attempt by the trap handler to look at the operand specifiers of the suspended instruction might itself cause a trap for reading a code-only segment (unless the trap handler modified the segment's access bits, the legality of which is unclear for a segment containing a suspended instruction). Of course, the trap handler could attempt to construct a MAP instruction containing the operand specifiers of the trapping instruction, but this technically constitutes self-modifying code, which cannot be executed implementation-independently (see "Cacheing and Pipelining", below).

- Whenever an I/O interrupt occurs, an arbitrary area of memory within the interrupted task may be undefined, since a string instruction might have been executing. Each subsequent interrupt has the potential of increasing the amount of memory with such implementation-dependent contents. A complete characterization of the system state -- in, for example, a memory dump -- may be very difficult, if not impossible, to understand.
- The above description of interruptability assumes that the trap handler will return using the RET instruction. What if ERET is used? Does the instruction remain suspended pending return from the Supervisor Exception Handler? Are new implementation dependencies introduced by this sequence of events?

The above-quoted section was added to the "Date TBD" version of MIL-STD-1862A to help clarify a number of questions raised during review of earlier drafts. However, *significant problems remain.*

3.3. Cacheing and Pipelining

Cacheing and pipelining are two "implementation speedup" techniques. In cacheing, certain recently- or frequently-referenced memory locations are maintained in registers or other fast-access storage, rather than *main memory*; in pipelining, sub-parts of one or more machine instructions are performed in parallel and/or in a time-optimal sequence, rather than in the order suggested by the architectural description. In conventional architectures, pipelining and cacheing are visible to the programmer in only a very few well-defined situations, if at all. For example:

In VAX family processors, the cache is implemented in such

> a way that its existence is transparent to software (except for timing and error reporting/control). [12, p.84]
>
> Each CPU may have an associated cache. The effects, except on performance, of the physical construction and the use of distinct storage media are not observable by the program. [9, p.14]
>
> In very simple machines in which operations are not overlapped, the conceptual and actual order [of storage references and instruction processing] are essentially the same. However, in more complex machines, overlapped operation, buffering of operands and results, and execution times wich are comparable to propogation delays between units can cause the actual order to differ considerably from the conceptual order. In these machines, special circuitry is employed to detect dependencies between operations and ensure that the results obtained are those that would have been obtained if the operations had been performed in the conceptual order. [9, p.23]

That is, conventional architectures go to some lengths to shield the programmer from the effects of implementation speedup techniques. However, regardless of assurances to the contrary (see the above references to "visibility" in [1]), many aspects of MIL-STD-1862A serve to place upon the programmer, rather than the hardware, the burden of avoiding cache- and pipeline-related anomalies. The following is a non-exhaustive list of such cases:

- In Section 8.1.3, "Cacheing of the Context Stack", MIL-STD-1862A states:

> In many implementations it will be desirable to maintain such information in fast registers... The Nebula architecture does <u>not</u> define the properties of any such cacheing mechanism. The representation of the context area of the active (Kernel and Task) context stacks is IMPLEMENTATION DEPENDENT. The value of such memory locations is undefined. The effect of storing into such memory locations is unpredictable. [3, p.25; emphasis in original]

- In Section 15.2, "I/O Space Assignments", MIL-STD-1862A states:

> Accesses to ALL registers in I/O space are restricted... Accesses that do not meet these restrictions shall produce one of two outcomes; either the access shall complete as requested or the access

> shall produce a hard memory trap. The choice is implementation dependent.
>
> The Kernel context pointer, the Task context pointer, the User map pointer, and the Supervisor map pointer are special registers that determine the control flow of the computer. As such, reading them through the I/O space may yield old or undefined values. Writing these registers through the I/O space will produce implementation dependent results. [3, p.71]

The Nebula restrictions on I/O space access are apparently due to implementation considerations. While such restrictions have no direct effect on portability, allowing an implementation to arbitrarily decide whether to enforce them is not reasonable. Nebula machines should all either trap on invalid I/O-space accesses or else perform them properly.

- In Section 12.3.1, "Cacheing of Memory Maps", MIL-STD-1862A states:

> In many implementations, it will be desirable to cache parts of the memory maps, such as the map size and a few recently used map entries. The properties of any such cacheing mechanism are implementation dependent. [3, p.47]

Subsequent paragraphs contain requirements that the LTASK and REPENT instructions "force the cache to be consistent" with the storage representation of all or part of the map. It is not at all clear, however, what this "consistency" entails, nor how long it lasts.

- In Section 12.2.5.1, "Self-Modifying Code", MIL-STD-1862A states:

> If access protection is disabled, it is possible to execute instructions that write their operands into the instruction stream in the immediate vicinity of the program counter... Since modern implementation techniques usually require some type of instruction pre-fetch, the action of such self-modifying code is unpredictable. Modifications (or data writes) to the instruction stream are guaranteed to be interpreted as stored only if a REPENT or LTASK instruction is executed before execution of the modified instruction

stream is begun. [3, p.47]

While it is true that restrictions on self-modifying code are not uncommon in "modern implementation techniques", the above specification is unsatisfactory in several ways:

- First, and most important, the term "immediate vicinity" is completely undefined! Self-modifying (whether intentionally or as the result of an error) programs will work on certain Nebulas without the use of LTASK or REPENT, but will fail on machines with a different "window of unmodifiability".
- Although self-modification is generally considered to be a poor programming practice, it must be recognized that the initial loading of a program into memory and any use of code overlays involve modification of program memory. To what extent are these operations legal, and what does the programmer *need to do* to ensure their proper function?
- In the System/370 architecture, consistency of the instruction pipeline and memory may be achieved by executing a particular form of no-op instruction. This is a small (2-byte), quick (no memory references), side-effect-free instruction that may be executed by any process in any state. By contrast, the LTASK and REPENT instructions involve significant CPU processing, require operand evaluation, may cause exceptions and traps, and need special privileges for execution. The request for instruction-stream consistency is very difficult.

- In Section 12.3.3, "Aliasing of Physical Addresses", MIL-STD-1862A states:

Using the relocation facility of the memory management

> system, it is possible to map two distinct virtual addresses onto a single physical address. This is known as aliasing of a physical address. In pipelined implementations, it may be desirable to use virtual addresses for data access coordination. In this case, the order of multiple accesses to the same physical address through different virtual addresses is unpredictable. The practice of aliasing physical addresses should be avoided. [3, p.48]

In many architectures, a prohibition against the sort of aliasing described here might be reasonable. In the case of Nebula, however, aliasing provides the only convenient way to implement storage that is readonly to a user but read/write to a supervisor. Some less sweeping elimination of aliasing is required.

3.4. The Procedure Interface

Nebula's procedure interface is one of its most distinctive features. A stack of "procedure contexts" maintains, in an implementation-dependent manner, the "current state of execution" [3, p.23] of all active procedures, including PSW, registers, parameters, exception handler, etc. By leaving the format of the context area open, the Nebula designers meant to provide "considerable freedom in the structure of the local store" [1, p.37]. As mentioned earlier, all aspects of this structure were to be invisible to the software.

In reality, however, the context stacks are not software-invisible. An operating system may choose to restrict a user program from accessing this memory, but there is no architectural requirement that it do so. MIL-STD-1862A does require that, at the time of call, the context area for the called procedure -- and perhaps the calling procedure, although this is not clear -- occupy context-only storage. For procedure

contexts further down the call chain, in suspended tasks, or in storage that has been re-mapped for any of a variety of reasons, however, there is no architectural reason for an operating system to restrict user access. And, of course, the operating system itself can clearly choose to examine or modify context-stack areas. Any such reference -- whether intentional or the result of a programming error -- may produce different results on different "Standard" machines, and therefore entails a portability problem.

Not only is it possible for software to access Nebula's procedure contexts, but in certain cases such access represents the only (or only convenient) way of performing a necessary function. Most of these involve anticipated requirements of operating-system or other supervisor-level routines. Although it may be possible to meet one or more of these requirements in a non-implementation-dependent manner, the burden of proof should reside with those making such a claim. "Proof", in this case, can only consist of a fully functional machine-independent operating system.

The unspecified format of Nebula's context stack presents problems in at least the following cases:

- The size of a procedure's context area will vary from implementation to implementation. Size information is needed in order for the operating system to allocate sufficient space for interrupt processing, as well as for the supervisor exception handler, trap handlers, etc. (See below for a list of unresolved issues in the area of context-stack overflow.) Although certain components of the procedure context strongly suggest storage sizes (e.g., the PSW and registers), the space requirements for others are intentionally left open by MIL-STD-1862A. In

particular:

- "The size and format of the parameter descriptors is implementation dependent" [3, p.32].
- "Encodings of the states [of the exception handler for a procedure] in the context area shall be implementation dependent" [3, p.35].
- "...The intermediate state of the instruction [suspended due to an interrupt or trap] is preserved (in the context stack, in an implementation-dependent form)" [3, p.129].

- The current Nebula procedure context is indicated by a pointer whose exact value is implementation dependent:

> The address in the context pointer shall be greater than or equal to the smallest address occupied by the current context. When a new context is created, the context pointer prior to being decremented shall be greater than the address of any byte of the newly created context. These restrictions imply that a context area may be initialized by setting the context pointer to the greatest word address in the context area plus 4. [3, p.25]

It is not clear that this provides enough information for an operating system to initialize context areas in all cases, and might easily lead to a wide variety of portability problems.

- Many programming languages provide a form of dynamic storage allocation for which periodic "garbage collection" is, if not absolutely required, at least desirable. Among the many ways of performing garbage collection, the most common involve "marking algorithms", which require that all currently-accessible data items be flagged. Data items may be accessible through pointers contained in registers and parameters of non-current procedures. A garbage collection routine must therefore have access to such registers and parameters, which in Nebula may be obtained only in

an implementation-dependent manner.

- Similarly, a run-time debugging package requires knowledge of register and parameter values for all procedures in a call chain in order to provide the programmer with a complete description of the system state. Again, only a machine-dependent version of such a package could be built in Nebula.

In addition to these format-related points, the following portability problems exist with respect to the calling mechanism itself:

- Except possibly for register 1, register contents upon procedure entry are undefined. Thus, accessing a register before assigning it a value will produce implementation-dependent results. Since "use before definition" is one of the most common programming errors, this is likely to be a major source of portability problems. An ISA Standard should either specify initial register contents or else stipulate that an Uninitialized.Value (or some similar) trap must occur upon reference to an uninitialized register.
- It is unclear what happens if there is insufficient storage in the current context-only segment at the time of procedure invocation. Will a contiguous context-only segment be used if it exists? If so, does the new context cross a segment boundary? What sort of re-mapping, if any, may a trap handler perform in such a case to expand the available context-only memory? What if there is insufficient context-stack space for the trap handler to be called? Similarly, what happens when insufficient context-stack space exists for processing an I/O interrupt, hard or soft memory error, exception, etc.?

One additional confusion about Nebula's context areas is worth

noting. On page 24, MIL-STD-1862A describes in detail the order of storage within a procedure context of the PSW, registers (even indicating the sequence in which these appear), parameter descriptors, and exception-handler state. Since at least one item is omitted from the list (the "intermediate state" of a string instruction suspended by a trap or interrupt), since the formats and lengths of the other items are machine dependent, since the item pointed to by the context pointer is also machine dependent, and since context-stack storage is not supposed to be software-visible in the first place, this structural information is of no use to the programmer. Its only apparent effect is to place constraints on the implementors, which is contrary to the general flavor of the "Standard".

3.5. Additional Problem Areas

In the previous sections of this report, the deficiencies in a few areas of the Nebula architecture were described in detail. The following list presents in more general terms some of the portability problems in the remaining areas of Nebula:

- **Sensitive Fields.** Nebula defines a great number of fields whose contents are implementation dependent, reserved, or otherwise restricted. Software modification of many of these fields produces "unpredictable" results. Every field of this type provides yet another opportunity for a program to act differently on different Nebula implementations.
- **Maximum Number and Minimum Sizes of Segments.** Nebula allows each implementation to decide on the maximum number of memory-management segments permitted (at least 16) and the minimum

acceptable size of each (at most 256 bytes). Since there appears no way for software to determine these maximum and minimum values, a transportable Nebula operating system that takes full advantage of hardware support for memory management would be impossible to construct.

- **Data on Segment Boundaries.** MIL-STD-1862A states

> When a 2, 4, or 8 byte primitive data object... is being accessed, segment association, relocation and protection checks function as if the object were referenced one byte at a time. [3, p.45]

For all of the cases in which memory is accessed other than for a primitive data object, how are segment association, relocation, and protection checks performed? These cases include at least the following: Instructions (including up to 257 operands on the CALL instruction and up to 65,539 on the CASE instruction), Procedure Descriptors, and SVC and OPEX Vector Tables.

- *Undefined Operand Sizes.* MIL-STD-1862A specifies that:

> If...an instruction encounters an operand whose size is not defined in the instruction description, the instruction shall abort and the PC shall be reset to the beginning of the instruction. An OPEX vectored call shall be initiated using the instruction's opcode as the index and its operands as parameters. The number of parameters for this OPEX procedure will be the same as the number of operands defined for the opcode in its instruction description. [3, p.171]

There is at least one case still left open by this provision: If the index specifier in scaled- or unscaled-index addressing mode is a 64-bit integer, the resulting address is "undefined". In other cases, the OPEX convention raises troublesome questions: For several Nebula instructions, the number of operands is determined by the programmer. How many are passed to the OPEX call? For CASE, LOOP, IBLEQ, and the various other instructions

whose operands include items not represented by one of Nebula's operand specifiers, how are the parameters to the OPEX call specified at all?

- **Debugging Facilities.** Nebula allows the programmer to "trace" execution, either on a statement-by-statement or procedure-by-procedure basis, by setting bits 13:14 of the PSW. Proper setting of these bits will cause a "break" to occur "after the execution of the specified instruction [all instructions or just procedure calls and returns] and before a check for pending interrupts" [3, p.39]. When a "break" occurs, control is transfered to the Supervisor Exception Handler. In concept, this is one of Nebula's nicer features. However, since no implementation details are specified, questions arise which, when answered independently by various Nebula implementors, will lead to transportability problems: What if an exception or trap occurs while processing an instruction that would normally cause a "break"? Will the break occur before or after processing the trap/exception, or not at all? Will this depend on the type of trap/exception?
- **Arithmetic Status Bits.** How are the C, Z, N, and T bits affected by traps and exceptions? In the case of the EDIV instruction, "if storage of either R1 or R2 is blocked by the memory management system, the storage is aborted and the operands are unaffected" [3, p.81]. Are the Z, N, and T bits unaffected as well? Does the same answer apply to all other instructions?
- **Sizes of Certain Parameters.** It is essential to the Nebula architecture that each parameter be associated with a <u>size</u> (1, 2, 4, or 8 bytes). MIL-STD-1862A therefore carefully specifies the

size of each of the "implicit parameters" to the Supervisor Exception Handler, etc. Apparently by oversight, however, sizes for the following are not given: the "priority level" passed as the single parameter to the software-interrupt-request procedure [3, p.40]; parameters 3 and 4 of the memory-management-trap procedure [3, p.48].

- **Illegal.Divisor.** For no apparent reason, MIL-STD-1862A is inconsistent in its treatment of the Illegal.Divisor exception, raising serious questions about how exceptions affect arithmetic instructions in general. For the DIV, REM, EDIV, and DIVU instructions, MIL-STD-1862A states: "When this exception occurs the operands are unaffected" [3, p.78, 80, 81, 88]. For the MOD and DIVFIX instructions, however, there is no such comment, and the description of the LOOP instruction actually implies that some operands may be modified even if an Illegal.Divisor exception occurs. In general, when does an exception prevent operand modification and when does it not? In what sense can two versions of Nebula be considered the same machine if they act differently in such cases?

PART 4 - CONCLUSIONS AND RECOMMENDATIONS

As a document, MIL-STD-1862A is not sufficiently clear, precise, or complete to be used as the definition of an ISA Standard. It allows such a wide range of software-visible variations that transportable programs will be difficult to write and impossible to certify as transportable. Detecting whether a given program intentionally or inadvertently utilizes any of Nebula's implementation dependencies is, in general, impossible.

The feasibility of a software-augmented execution environment that simulates an implementation-independent "virtual machine" remains to be demonstrated. It is questionable whether such an environment can be constructed, however, and almost certain that it would have to be re-constructed for each Nebula implementation.

MIL-STD-1862A could be enhanced by adherence to the following guidelines:

1) The operation of the machine in all cases must be unambiguously specified by the Standard, even if implementation dependent. That is, the Standard must be complete.
2) All implementation dependencies should be made explicit in the Standard and the range of variability made clear.
3) A design philosophy should be adopted in which implementation dependency is the exception, rather than the rule. This requires, at least initially, a completely specified Standard with no implementation dependencies.
4) A consistent set of principles must be developed and applied in determining what implementation dependencies will be allowed. These principles should involve primarily considerations of

costs and benefits, including hardware and software costs for the full life cycle of a standard machine. Major elements of the cost/benefits analysis include:

- How much faster and/or cheaper will a machine be if it can take advantage of the flexibility provided by the implementation dependency?
- To the extent that the implementation dependency is software-visible, how difficult will it be to detect a given program's inadvertent or intentional use of it? Can a compiler or operating system detect such use? How expensive will it be to put such detection in the compiler or operating system? Will such an addition make the compiler or operating system itself non-transportable?

5) Proposed implementation dependencies should be presented to a control board for evaluation according to the principles mentioned in (4). Approved implementation dependencies must be carefully and completely documented, including a description of the techniques available for detecting improper use.

The dilemma facing Nebula is a classic one: How to take advantage of advances in a rapidly advancing field, while allowing "old" programs to continue working across a wide range of systems. Although well-intentioned, the current Nebula approach to this dilemma has resulted in an under-specified Standard in which the burden of writing transportable software rests with the programmer. Both DEC and IBM have proven that there are better approaches.

REFERENCES

[1] Szewerenko, Leland, William B. Dietz, and Frank E. Ward, Jr., "Nebula: A New Architecture and Its Relationship to Computer Hardware", Computer 14 #2 (February, 1981).

[2] "Electronic Systems Division Plan for High Level System Standardization Program", Electronic Systems Division, Air Force Command, February 9, 1981.

[3] Instruction Set Architecture for the Military Computer Family, MIL-STD-1862A, Date TBD.

[4] Burr, William E., Samuel H. Fuller, and Harold Stone, "Computer Family Architecture Selection Committee -- Final Report, Volume II -- Selection of Candidate Architectures and Initial Screening", Report ECOM-4527, U. S. Army Electronics Command, Fort Monmouth, NJ, September, 1977.

[5] Padegs, Andris, "System/360 and Beyond", IBM Journal of Research and Development 25 #5 (September, 1981).

[6] Wulf, William A., "Compilers and Computer Architecture", Computer 14 #7 (July, 1981).

[7] IBM System/360 Principles of Operation, Order No. GA22-6821, IBM Corporation.

[8] Bhandarkar, Dileep, "Architecture Management for Ensuring Software Compatibility in the VAX Family of Computers", Computer 15 #2 (February, 1982).

[9] IBM System/370 Principles of Operation, Order No. GA22-7000, IBM Corporation.

[10] VAX11 Architecture Handbook 1979-80, Digital Equipment Corporation.

[11] VAX Software Handbook 1980-81, Digital Equipment Corporation.

[12] VAX Hardware Handbook 1980-81, Digital Equipment Corporation.

[13] Clearwaters, Allan, Robert Gordon, and Daniel Siewiorek, "Computer Family Architecture Selection Committee -- Final Report, Volume VIII -- CFA Final Selection", Report ECOM-4531, U. S. Army Electronics Command, Fort Monmouth, NJ, September, 1977.

Analysis of Nebula Architectural Support for I/O

Robert D. Cowles

December 3, 1981

Digicomp Research

Ithaca, NY

References to Nebula in this report refer to the version of MIL-STD-1862A as changed through 31 August 1981

Analysis of Nebula Architectural Support for I/O

Robert D. Cowles

Digicomp Research Corporation

Ithaca, NY 14850

December 3, 1981

The Input/Output subsystem of a computer instruction set architecture must be considered at least as important as the design of central processor instructions. The I/O interface directly affects such architectural goals such as program portability, security, program verification, performance (both for multiprogramming and real-time applications), and fault tolerance. The purpose of this paper is to examine the Nebula I/O interface as specified by MIL-STD-1862A, with modifications through 31AUG81. The first section presents a summary of the I/O interface with emphasis on those areas of concern to a programmer who must understand the relationships between the Central Processor, I/O Processors, and Memory Management Subsystem. The second section details areas of the I/O interface specification that are ambiguous or problematic in terms of meeting the architectural goals of Nebula mentioned above. The third section discusses some possible approaches which could be used by desig-

ners of operating supervisors for Nebula computers which could overcome some of the problems. The fourth section proposes some changes to MIL-STD-1862A which, if implemented, would eliminate certain problem areas. The fifth section considers the relative merits of changing the Nebula standard against the potential impact of the problem addressed.

THE NEBULA I/O INTERFACE

Introduction

The major part of MIL-STD-1862A dealing with the I/O subsystem is concerned with interfaces between the central processor (CPU) and special purpose processors known as I/O Controllers (IOCs). The standard IOC has a limited instruction set but can execute independently of the CPU and provide a relatively simple interface to a variety of devices.

For the most part, the IOC is assumed to be connected to devices by one of three different interfaces: Parallel Point-to-Point (PPP) provides a high speed 16 bit connection to a single device; Serial Point-to-Point (SPP) provides a slow speed interface to a device using protocols like RS-232; and MIL-STD-1553B (1553B) provides a 16 bit bus connection between a variety of devices and is widely used in military applications. An attempt has been made in Nebula to provide a single architecture containing enough flexibility to utilize the capabilities of all three types of interfaces.

Physical Memory Address Assignments

I/O Space: The first megabyte of the physical address space is set aside for use as I/O and CPU control registers. The upper 2K bytes of

I/O Space is reserved for such processor control registers as: PSW, ASR, map pointers, context pointers, timers, and OPEX and SVC vector pointers and limits. The remainder of the I/O Space is used to access or control either devices or IOCs. Each IOC is allocated a 512 byte block of registers in I/O Space aligned on an address that is a multiple of 512.(1)

Because the implementation of this portion of the address space is likely to be much different than program memory, in I/O Space the CPU is not allowed to generate references which cross register or data item boundaries. Also, no IOC program, message, or data access is allowed to I/O Space (access causes a hard memory error or a Memory.error interrupt from the IOC).

The lower 256 bytes of the IOC register block is for secure data which untrusted programs should not be allowed to access. Currently defined registers in this block are:

1. Channel Configuration Register: contains interrupt priority for attached to the IOC. The register may also contain channel dependent information like baud rate, Remote Terminal (RT) address (for 1553B serial bus interface), or device interrupt priorities (for parallel point-to-point interface).
2. Program Segment Specifier: sixteen bytes containing sufficient implementation dependent information such that the IOC may verify that a channel program instruction or literal access is within the

(1) The block size of 512 bytes was chosen because some implementations may use 256 for the minimum memory segment size. It was desired to have two portions of the block be independently mapable so a user process would not have to be given access to the secure portion of the register block (see below) but would still be able to perform direct I/O operations to a device.

original segment specified with a SETSEG instruction and to perform the necessary relocation of virtual addresses associated with the channel program.

3. Message Segment Specifier: similar to the Program Segment Specifier except that it is used to validate/relocate contents of and offsets from the Message Pointer Register (below).
4. Data Segment Specifier: similar to the Program Segment Specifier except that it is used to validate/relocate virtual addresses for IOC instructions that transfer data (except for instructions that specifically use the message segment for data transfer).

The upper half of the IOC register block contains:

1. Channel Status: sixteen bits with bits 2 through 15 containing channel dependent information. Bits 0 and 1 control starting and stopping the IOC and are discussed later.
2. Channel Program Status: sixteen bits used to indicate the reason for early termination of a data transfer (Overrun, Data Check, etc.).
3. Channel Program Counter: contains the 32 bit virtual address of the next instruction to be executed.
4. Message Pointer: contains the 32 bit virtual address of the current message. In simple cases, the message contains the virtual addresses of data buffers to be used for data transfer operations.
5. Status Word and Vector Word: sixteen bits of flags defined for use by the 1553B interface.

Other assigned Physical Addresses: Certain physical addresses of interest are located in the first 256 bytes above the I/O Space (starting at

hex address 00100000). The reserved addresses in this section of memory should be accessed relatively infrequently so that little performance impact results from having the values in memory rather than in registers in I/O Space.

Reset/IPL entry and save area pointers: The entry address of the procedure to be called when the RESET switch is activated is located at hex location 00100040. Location 00100044 points to a doubleword which contains pointers to the supervisor map and kernel context area to be used when the procedure is invoked. The IPL process is similar to the RESET sequence except that the procedure entry address is provided by the loaded IPL text.

Device Interrupt vectors: Locations with hex addresses 00100060 through 001000FF are reserved for device interrupt vectors. Each IOC is assigned a four word interface dependent interrupt vector of the format shown in Figure 1 below. Note that at a minimum, each type of interface has a separate vector address for program interrupts (generated by the IOC instruction INT) and for IOC error interrupts. The procedure invoked has at least one parameter which is the physical address of the interrupt vector that was used.

Central Processor interaction with I/O

The Central Processor interacts with the I/O subsystem through the sharing of memory (described above), special instructions, access to IOC registers, and interrupts. This section discusses the latter three areas of interaction.

Location	Parallel	Serial PP	1553B
001000x0	program	input program	program
001000x4	error	input error	error
001000x8	device	output program	reserved
001000xC	reserved	output error	reserved

Figure 1: Interrupt Vector Assignments

SET I/O Segment - CPU instruction: This instruction has two address operands, a Seg operand that maps to the physical address of an IOC segment specifier and an Adr operand that specifies a virtual address in the segment to be used. If the Seg operand does not map to an IOC segment specifier, a Segment.specifier exception or hard memory error will result. If protection is enabled, the protection attributes of the segment containing virtual address Adr are checked. A channel program segment must have instruction access; message and data segments must have read/write access. An invalid Adr operand causes condition code bits to be set indicating the type of error (Z is set for invalid address, N is set for protection violations) and the segment specifier is set to prohibit all accesses. To enable an operating supervisor to easily prohibit all accesses, the virtual address FFFFFFFF is always considered invalid. As mentioned before, the contents of the IOC segment specifiers are implementation dependent but sufficient to allow an IOC to validate a virtual address as being within the specified segment and to relocate the virtual address to a physical address.

Starting and Stopping I/O: The process of starting an I/O operation on an IOC is relatively simple. In the general case, the procedure to start an I/O operation requires the following addresses:

1. Pointer to the first instruction in the Channel Program. This address also defines the channel program segment.
2. Pointer to the Message area. This address also defines the message segment.
3. Pointer to the data buffer. This address also defines the data segment.
4. Pointer to procedure to be invoked in case of an IOC error interrupt.
5. Pointer to procedure to be invoked by an IOC program interrupt.
6. Pointer to the IOC register block, or some indication of the IOC which is to perform the operation.

Using this information, the following steps can be performed to initiate an I/O operation:

1. Verify, using bits 0 and 1 of the Channel Status Register for the IOC that the IOC is not active. The first two bits of the Channel Status Register indicate the current overall status of the IOC. The CPU sets/clears bit 0 to request the IOC to start/stop the current channel program. The IOC sets/clears bit 1 of the Channel Status Register to indicate whether it is currently active/stopped. If the IOC is active when the request is made, the program may want to: 1) queue the request; 2) halt the current I/O operation; or 3) pass some error indication back to the caller.

2. Issue the SETSEG instruction for the three segments. If the SETSEG fails for any of the segments, the program may wish to return an error indication or it could continue and allow the IOC error interrupt to be taken when the IOC discovers that no access is allowed to the segment.
3. Place the pointers to the interrupt handling procedures in the appropriate interrupt vectors (or in a control block accessible to system provided interrupt routines).
4. Place the Channel Program address and the Message address in the Channel Program Counter and Message Pointer of the IOC.
5. Set bit 0 of the Channel Status Register indicating that the IOC should start.

Note that only steps 2 and 3 of the above process require intervention by a privileged procedure. Once those steps have been performed, a non-privileged process with access to the upper half of the IOC register block could, with reasonable security, be allowed to start and stop its own I/O operations.

CPU interrupts: An IOC or device may interrupt processing by the CPU if the priority of the interrupt request is greater than the priority at which the CPU is currently executing (contained in the PSW). Interrupts are treated as procedure calls with the device or IOC (plus IOC interrupt type) determining the fixed location in memory containing the procedure entry address. All interrupts execute on the Kernel context stack and the Base bit of the current context is set on. The interrupt procedure will be considered privileged if the entry in the interrupt vector pointing to the procedure has bit 31 set on.

If a device or controller requests an interrupt, it specifies the interrupt priority (in the range 0-31) and the physical location of the interrupt vector containing the virtual address of the interrupt procedure. The procedure has access to the physical vector address as a single read-only parameter.

IOC requested interrupts may be generated by errors or by the INT instruction. The procedure invoked is specified in the error or program interrupt elements of the four word interrupt vector assigned to the IOC (See Figure 1). When invoked the procedure has access to two parameters: the physical vector address which contained the address of the procedure; and the interrupt code.(2) The channel program may be suspended if an INT request cannot be serviced because the CPU is executing at an equal or higher priority.

I/O Processor Instruction Set

In general, IOC instructions are halfwords and are classified as data transfer instructions or control instructions. The 16 bit instructions contain an eight bit operation code and an eight bit value that is usually interpreted as an offset or index in halfwords from the value in the message pointer register. For the PPP interface, bit 7 of the operation code indicates that the information to be transferred is data (bit 7 = 0) or control/status (bit 7 = 1).

IOC Transfer Instructions: At the start of a data transfer operation the accumulator contains the number of units of information to transfer. At the end of the operation the accumulator contains the number of units

(2) The interrupt code is the IOC error code for interrupts caused by IOC errors. For interrupts caused by the INT instruction, the interrupt code is the value of the IOC accumulator.

in the original count that were not transferred (the channel program status register contains the reason for early termination of a transfer operation). For the PPP and 1553B interfaces, a unit of information is a 16 bit halfword and for the SPP interface a unit of information is an eight bit byte.

The following section briefly describes the most general cases of the IOC data transfer instructions. Instructions used with specific interfaces may differ slightly in detail or in functions performed.

* READ - specifies offset in message that is a 32 bit virtual address of a buffer. The buffer must be in the data segment.

* RDTMSG - specifies offset in message that is the starting location to receive the information.

* WRITE - specifies offset in message that is a 32 bit virtual address of a buffer containing that data to be written. The buffer must be in the data segment.

* WRFMSG - specifies offset in message that is the starting address of the data to be written.

* WRLIT - the offset is a reserved field. The instruction is followed by a 32 bit virtual address of the data to be written which is in the program segment.

* RT2RT - index value into message is a four halfword block used to initiate transfer between two remote terminals on a 1553B interface and to store status information at completion of the transfer.

IOC Control Instructions: This section briefly describes the IOC control instructions. The description is designed to give an idea of the kind of instructions available. These instructions operate consistently for all types of interfaces.

* LOAD - load accumulator from offset into message.

* STORE - store accumulator at offset into message.

* LOADST - load accumulator from offset into IOC register block.

* IADD, ISUB, IAND, IOR - perform the operation with the accumulator and the indexed halfword from the message. Place the result in the accumulator.

* LOADL, IADDL, IANDL, IORL - perform the operation with the accumulator and a 16 bit literal following the instruction. Place the result in the accumulator. The last eight bits of the LOADL and IADDL instructions are reserved.

* ADDTA - indexed location in message specifies a 32 bit word in the message that is added to the contents of the accumulator and replaced by the result.

* LSHFT - the accumulator contents are shifted left or right by the amount specified in bits 8 through 15 of the instruction (interpreted as a signed shift count).

* LMP - indexed location in message is a 32 bit virtual address which replaces the current contents of the message pointer register. If the new message pointer is zero, the program is halted and bits 0 and 1 of the channel status register are cleared.

* BRIO, BRNEIO, BLSSIO, CASEIO, BCASE - transfer control within the channel program based on the contents of the accumulator. BRIO causes a branch regardless of the contents of the accumulator.

* INT - a CPU interrupt is generated at a priority as specified in bits 11 through 15 of the instruction. An IOC error interrupt of Interrupt.priority occurs if the specified priority exceeds the maximum priority specified in the channel configuration register. The contents of the accumulator will be passed as a parameter to the invoked procedure.

* HALT - equivalent to loading the message pointer register with zero.

* CONTROL - provides ability to perform interface dependent control functions for SPP and PPP interfaces.

I/O PROBLEM AREAS

Minor Errors or Omissions

The Nebula specification has suffered some consistency problems as changes are made that are not reflected in all parts of the document. For instance, although a set of corrections has been issued since the change to the size of the IOC register block, the description of the

LOADST instruction for the IOC has not been updated to reflect the new manner in which it must operate. Also, the description of LOADL and IADDL indicates that the last eight bits of the instruction are reserved; the description of IANDL and IORL contains no mention of such a restriction although there is no reason to believe the instructions are not similar in that respect.

The IOC HALT Instruction

The function of the HALT instruction seems adequately performed by the LMP instruction with a new message pointer of zero. Is the HALT instruction for the IOC necessary?

The SETSEG Instruction

The description of the SETSEG instruction misses several important details. The problems became apparent while thinking about the sequence of operations necessary to start and stop I/O, and the possible error conditions that would have to be handled.

Operand Access: The two operands of the SETSEG instruction are described as "address operands" yet one of the operands must map to one of the IOC segment specifiers in an assigned IOC register block. Since address operands do not cause access violations, is it really the designers' intent to allow a SETSEG instruction to be issued for an IOC whose register block was covered by a virtual segment with "no access" protection? An alternative interpretation which would require "read/write" access seems to eliminate much of the protection afforded by the SETSEG instruction. Regardless of the interpretation, the term "address operand" is being used in a fashion which is not entirely consistent with

the spirit of the rest of the specification and a more detailed explanation is warranted.

IOC Active: The description of the SETSEG instruction contains no restrictions on issuing the instruction while the IOC is active. In other sections of the specification, an IOC error of IOC.active is generated if registers such as the channel program counter are altered during IOC execution. Is it an oversight that an IOC error interrupt is not specified when a SETSEG is executed for an active IOC?

Maximum Priority for INT Instruction

The generation of an IOC error interrupt when the priority in the INT instruction exceeds the maximum priority seriously affects program portability. The maximum channel program priority is contained in the channel configuration register. Programs using the IOC cannot easily access the maximum priority since access would not normally be given to the lower half of the IOC register block. The priority of a program interrupt is contained in the INT instruction which is in a segment with "instruction" access and is therefore not easily modified. If the INT instruction specifies a priority larger than the maximum priority, an IOC error interrupt occurs. A program cannot find out what the maximum priority is and would, at any rate, have difficulty changing the priority in the INT instruction.

Device and IOC Interrupt Vectors

The ability of an operating supervisor to be flexibly configured for a particular I/O configuration brings up some problems with the manner in which the interrupt vectors are specified in Nebula.

Limitation on Number of IOCs: The area reserved for device and IOC interrupt vectors is from location 00100060 through 001000FF. This address range provides for a maximum of ten I/O Controllers to be attached to a Nebula computer. While the maximum is probably sufficient for many embedded computer applications, ten IOCs is a serious restriction if the Nebula architecture is ever to be extended to mainframes. Note that any directly connected devices have to be provided with interrupt addresses further reducing the number of IOCs that may be attached.

Program Visibility of Interrupt Vector: In many circumstances, the operating supervisor will need to know the address of the interrupt vector associated with a particular IOC or device. Trusted programs which are allowed to perform their own I/O operations and field their own interrupts will need a method of specifying procedure entry addresses in the interrupt vectors. The only safe method is to supply this information to the operating supervisor which fills in the appropriate interrupt vector -- so long as the supervisor knows the location of the interrupt vector to be used. Requiring the operating supervisor to be pre-generated with the location of the interrupt vectors for each IOC greatly restricts portability of the supervisor and seems unnecessary since the IOC must have the vector address available internally.

Reset and IPL Sequences

The Reset and IPL sequences require more specification. It is crucial that the machine be in a specific state when the reset or IPL routines are invoked. Examples of areas of concern are:

* The state of the IOCs and devices following a reset or IPL are not specified ... are they halted or might they be active?

* New kernel context stack and supervisor map pointers are contained in two words pointed to by location 00100044 ... exactly when in the IPL sequence are these values made current?

OPERATING SYSTEMS DESIGN REQUIREMENTS

Impact of Previously Discussed Problems

Many of the problem areas discussed in the preceding text do not seriously affect the design of operating systems for Nebula computers because the problems relate to ambiguities or lack of detail rather than design flaws. Careful design of the operating system is required to meet goals of performance, security, verifyability, etc., but except for the problems outlined above and the discussion of I/O interrupt procedures discussed below, few serious obstacles to this process are seen. The problems of the maximum priority for the INT instruction and the limitation on the number of IOCs are of such a nature that there is no clear approach to the design of an operating system that would avoid them.

The remaining problem which can be partially solved is the one of program visibility of the interrupt vector addresses. The first thing to note about the problem of the vector addresses is that the interrupt procedure is passed the physical vector address when the information it really needs is which IOC generated the interrupt and why (error or pro-

gram interrupt). The "solution" to the problem then is to build a cross reference table between the physical vector addresses and the location of the IOC register blocks. This table must be regenerated each time the location of the interrupt vector is changed and the difficulty of finding errors in this generation process can be time consuming.

I/O Interrupt Procedures

I/O interrupts (and software interrupts) are much different than traps and exceptions because the task currently executing may have nothing to do with the reason for the interrupt. The interrupt is processed by switching to the kernel context stack and calling a procedure whose address is obtained from a certain physical location. The procedure must be present at the same location in the memory map of all tasks, so in most cases the I/O interrupt procedure will have to be referenced through the supervisor map. In a multi-tasking environment, a process which has been given the ability to initiate I/O operations by directly manipulating the registers in the upper half of the IOC register block cannot be allowed to handle I/O interrupts from the IOC without impacting security. Intervention by the operating system is necessary to perform a switch to the task responsible for the I/O before an application's I/O interrupt routine can safely be given control.

The task switch with associated loading and storing of cached information may impact the ability of a Nebula processor to respond quickly to interrupts, particularly if the amount of cached information becomes large in an attempt to improve processor speed. To avoid this impact, the operating system would have to provide a more complex (and unfortu-

nately, more programming error prone) interface to allow a validated supervisor procedure to handle I/O interrupt processing without requiring a task switch each time.

PROPOSED MODIFICATIONS TO NEBULA STANDARD

Clarifications

LOADST: The description of the LOADST instruction of the IOC should be changed to reflect the increased size of the IOC register block. The suggested change would be to interpret bits 8:15 of the instruction as a halfword offset from the CCR with bit 8 forced on to prevent access to the lower half of the IOC register block.

IORL and IANDL: A sentence should be added to the description of the IORL and IANDL instructions specifying that bits 8:15 of the halfwords containing these instructions are reserved.

SETSEG

Use of address operands: The impact of using address operands for the segment specifier requires more explanation. The suggested change is to specify that there must be a read/write segment mapping the lower part of the IOC register block and that protection can be provided by setting the privilege bit in the map entry for that segment.

IOC Active: The IOC is unlikely to perform properly if a SETSEG instruction is issued for one of its segment specifiers while it is active. Suggest the addition of a sentence specifying that execution of a SETSEG instruction by the CPU while the IOC is active (bit 1 of channel status set) shall cause an IOC error interrupt with fault code of IOC.Active.

Maximum Priority for INT

To allow increased program portablity, the INT instruction should be changed so that if the priority specified in bits 11:15 of the instruction exceeds the maximum priority in the channel configuration register, the maximum priority is used. References to an Interrupt.priority error interrupt for the IOC may then be deleted.

Device and IOC Interrupt Vectors

Problems of limitations on the number of IOCs and program visibility of the interrupt vector would be eliminated if the following changes were made:

1. When the IOC requests an interrupt, it provides the address of the IOC register block, the offset in the interrupt vector block to be used, and the interrupt priority.
2. The lower half of the IOC register block contains the 32 bit physical address of the interrupt vector to be used.
3. During interrupt processing, the CPU fetches the interrupt vector address and adds the offset to get the address containing the interrupt procedure entry address.
4. The interrupt procedure has access to an additional parameter which is the address of the IOC register block for the IOC requesting the interrupt.

The changes described above are meant to be as compatible as possible with the current Nebula architecture and are patterned after the processing for SVC and OPEX instructions.

Reset and IPL

The state of the machine after the reset or IPL switch is activated must be completely specified. In particular, the standard should include the stipulation that all IOCs are halted. Also, to prevent confusion, the standard should be more detailed in its description of the IPL sequence; i. e. there should be a sentence stating that the pointers to the kernel context stack and the supervisor map are loaded into their respective hardware registers after the IPL data is loaded but before the control is transferred to the procedure address specified in the IPL data. Any restrictions on the format of the IPL data should appear in the standard.

IMPACT OF MODIFYING THE CURRENT STANDARD

Considering the extent of the recent redesign of the IOC part of 1862a, it is no surprise that there are some additional problems which need to be addressed. Adoption of the proposed changes and adding some of the missing details to the standard should greatly improve confidence in the portability of operating systems between Nebula computers.

If the proposed changes or similar changes are not made to the Nebula standard, it will be quite possible that operating systems will not be transportable between machines that conform to the standard. Since the operating system is a critical interface between application programs and the hardware, requiring operating system changes to use different Nebula computers greatly increases the probability that the "environment" the operating system provides is not the environment required for the application program to operate correctly.

The changes proposed in the preceding sections have their primary impact on the IOC. Considering the recent changes to the IOC part of the specification (since the release of 1862a), it appears unlikely that many of the proposed changes would have a significant additional impact on the implementation effort. The major change suggested was in the IOC interrupt vectors; for that change, some care was taken to propose a scheme compatible with current handling of OPEX and SVC instructions so that common logic or microcode could be used.

JOVIAL/Nebula Suitability Report

Software Engineering Associates

October 6, 1981

References to Nebula in this report refer to the version of MIL-STD-1862A dated 1 July 1981.

Jovial/Nebula Suitability Report 10/6/81

1. Introduction

Nebula (MIL-STD-1862) is a 32-bit Instruction Set Architecture which is being developed for use in embedded computer systems. JOVIAL-J73 (MIL-STD-1589B) is a high-level language in which programs are written for these embedded computers. In this report we evaluate the suitability of Nebula as a target for applications which are written in JOVIAL. Any unqualified references to JOVIAL refer to the language which is described in MIL-STD-1589B; references to other JOVIAL dialects are qualified.

1.1 Background

Nebula was examined from three basic viewpoints:

1. With respect to efficiency of execution for JOVIAL programs.
2. With respect to ease of compilation into efficient code.
3. With respect to ease of compilation.

Note that these objectives are related, but are different. There may be an architecture for which it is possble to hand-code efficient programs but which does not lend itself to easy compilation. Similarly, a simple architecture may make it easy to compile code but difficult to generate efficient code.

It should be emphasized that not only must it be possible to write efficient programs for a given architecture, but it must also be possible for the compiler to translate source into efficient object programs without an undue amount of effort. An architecture for which a great deal of analysis is necessary to generate efficient programs may be worse in practice than another which is worse in theory, but whose potential is more easily realizable.

Problems which are encountered in typical JOVIAL implementations are discussed, roughly in order of decreasing difficulty of implementation on Nebula. In some cases problems which are, in general, potentially great for other architectures are relegated to the end of the list, because they pose no particular difficulties on Nebula.

Areas examined were:

1. Parameter procedures.
2. Truncation and rounding.
3. Parameter labels.
4. Abort statement.
5. Parameter passing and referencing.
6. Data referencing and storage allocation.
7. Bit strings.
8. Character strings.
9. Operand sizing.

10. Optimization.
11. General Machine Idiosyncracies.
12. Loops.
13. Case.
14. Part-Word Operands.
15. Star tables.
16. Fixed-point Arithmetic
17. Data Allocation.
18. Relationals in Value Contexts.
19. Tight Tables
20. Address Computations (including subscripting).

Several other areas, which have caused problems in the implementation of other languages, but which are of no great concern in the implementation of MIL-STD-1589B JOVIAL are also discussed. These include:

1. Parallel Tables.
2. Checking.
3. Tasking.
4. Input/Output.

1.2 Executive Summary

It is obvious that the Nebula Instruction Set Architecture has been designed with high-level languages in mind. On the whole, it is better suited for the implementation of JOVIAL than are most architectures. Among the features of Nebula which are particularly well-suited to JOVIAL are:

1. Fixed point operations.
2. Three operand arithmetic operations.
3. Associating sizes with operands, rather than with operators.
4. Relational operators which generate boolean results.
5. Scale operator.

Areas of Nebula where we see problems or inefficiencies are:

1. Parameter labels
2. Parameter passing and referencing
3. Data referencing and storage allocation
4. Parameter procedures
5. Character strings
6. Bit strings
7. Truncation and rounding
8. Operand sizing
9. Part-word operands
10. Optimization

11. General Machine Idiosyncracies

Parameter labels present some serious implementation problems, because the context stack is hidden from user programs. Additional instructions to make the manipulation of the context stack simpler are recommended.

The problems of parameter passing have to do with passing parameters by value and referencing up-level parameters. Changes to facilitate these operations are recommended.

Storage allocation is largely unsupported in Nebula. We feel that high-level support for allocating data on the stack should be provided.

Operations on variable length character strings which require padding present problems on Nebula. The JOVIAL rules for overlapped moves exacerbate the problems. A suggestion for additional character string instructions is made.

Nebula provides good support for short bit strings, but less support for longer ones. The addition of a long bit move is recommended.

JOVIAL associates rounding and truncation with individual operands, but Nebula has global flags which indicate what sort of rounding is to take place. Also, JOVIAL allows rounding and

truncation for integer and fixed-point items. Two alternatives for handling these problems are suggested.

JOVIAL permits part-word operands to be used in expressions. Nebula provides facilities for extracting and depositing these operands but does not allow them to be used directly in computations. We suggest a method for permitting this.

Features of Nebula such as the multitude of addressing modes, three operand arithmetic instructions, and the fact that registers are not passed to the callee, require new strategies for optimization if full advantage is to be taken of the architecture.

Nebula is better that most architectures in avoiding machine idiosyncracies. There are some minor ones, however, which are pointed out in this report.

It should be noted that in each of these areas Nebula is no worse than ordinary general purpose register architectures, and is better than most. Since we are examining the architecture before the machine has been built, we have used a stricter standard for judging than we otherwise would have to measure the "goodness" of the architecture. More high-level support is expected from Nebula than would normally be expected from an existing architecture.

2. Present JOVIAL Implementations

Before we examine in detail the problems of implementing JOVIAL, we discuss different architectures and the kind of support they provide. JOVIAL compilers have been implemented for a variety of target machines. MIL-STD-1589B compilers exist for the DEC-10, IBM370, TI990, 1750A (MIL-STD-1750A) and the Z8002, and are being developed for a number of other machines including VAX, and the PDP-11. Compilers for other dialects of J73 (MIL-STD-1589 and MIL-STD-1589A) exist for the Univac 1108 and Collins CAPS, among others.

Of all of these implementations, the CAPS architecture stands out as the one which supports JOVIAL most efficiently with respect to compactness of code. The MIL-STD-1750A is also relatively good. The machines with the larger word sizes (DEC-10, Univac 1108, and IBM370) do not fare as well.

There are several reasons for this. Most important is the efficiency with which it is possible to access operands. CAPS and the 1750A both provide shorthand encodings for operand addresses. CAPS is a stack-oriented architecture which addresses instructions by byte, but data by word. Since it is stack-oriented, operands are somewhat divorced from operators. A single byte instruction can be used to access any of the first 16 words of local data in a procedure. Two-byte instructions can address the first 256 words of data. Page registers are supplied

so that the references to commonly used data can be shortened. The 1750A architecture contains base registers and allows certain operations on dedicated registers and an operand within 256 words of the address contained in a base register, to be specified in only two bytes. The machines with the larger word sizes do not provide this sort of optimization. Memory reference instructions always take a full word or more.

Another area in which CAPS excels is that of procedure calls. Parameters are pushed onto the stack, just as for ordinary arithmetic operations. When a procedure is called, the local frame pointer is set up so that the parameters are in the callee's local frame. In the called procedure, parameters may be referenced as if they were local data. Upon return from a procedure, the requisite bookkeeping to restore the caller's context is performed by the hardware.

3. Nebula vs. Other Architectures

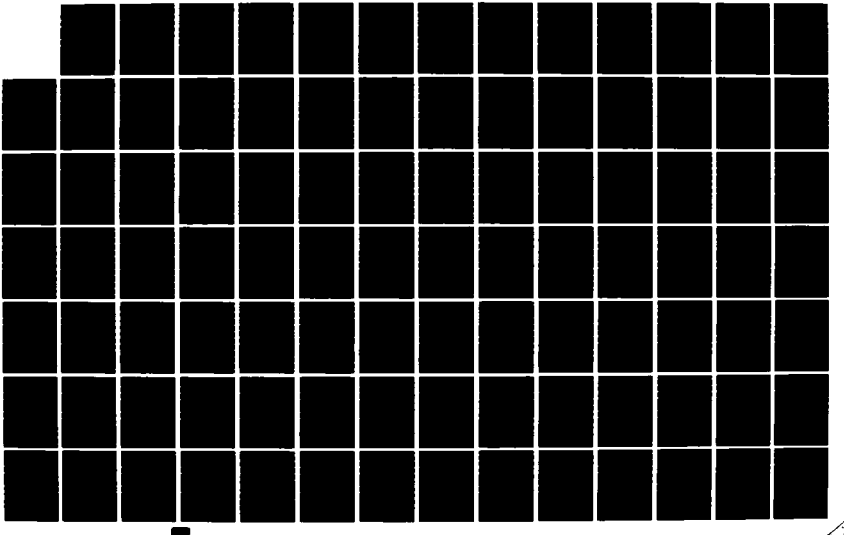
Since Nebula is an Instruction Set Architecture, rather than a machine, it is not meaningful to speak of its efficiency with respect to speed. Different implementations may have performance characteristics which vary considerably. It is meaningful to discuss compactness of code, however. As well as having a bearing on how much memory is required for a given application, it provides some clue as to how fast an implementation would be relative to comparable implementations of different architectures, since memory bandwidth is often a limiting factor in machine speed.

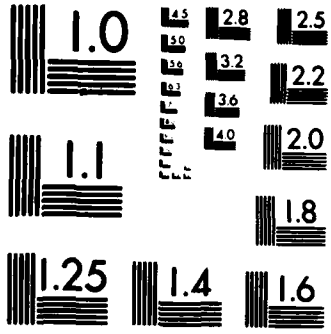
The Nebula architecture is more regular that either CAPS or the 1750A, but in some respects it is also less efficient for implementing JOVIAL. Both CAPS and the 1750A gain an advantage in terms of space because they use assumed operands. On CAPS only operands which are referenced from memory need be specified, because it is a stack architecture. On the 1750A the dedicated registers need not be specified, and base registers are restricted to a subset of registers. This makes for a compact encoding for some common operations.

CAPS is also more efficient with respect to procedure calling and parameter passing, than is Nebula, because value parameters and space management are both handled automatically. Nebula's parameter passing mechanism is more efficient that the 1750A's.

AD-A151 041 NEBULA INSTRUCTION SET ARCHITECTURE (ISA) EVALUATION 3/4
(U) DIGICOMP RESEARCH CORP ITHACA NY R D ARNOLD ET AL.
SEP 84 RADC-TR-84-190 F30602-80-C-0279

UNCLASSIFIED F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Both of these architectures suffer from the fact that value parameters must be explicitly copied into local space, although for small procedures, value parameters could be kept in the 1750A's registers.

Nebula is more efficient than the two architectures mentioned with respect to operands which require complicated address calculations. In both the CAPS and the 1750A the calculations must be done explicitly, while in Nebula they are part of the operand addressing mechanism.

Nebula is also more efficient for handling large programs than either of those architectures. CAPS has a somewhat awkward universal addressing scheme which must be used to reference code or data which is not in the same 64K.

Architectures such as the DEC-10 and the IBM 370 are not nearly as good as Nebula for compactness of code due to several factors. Nebula provides short address forms for data operands, and permits branches to be made relative to the program counter. Operand references and short relative branches occur often in computer programs, so the savings provided by short forms is significant.

Another advantage that Nebula has over these two architectures is that it provides more memory-to-memory operations. Since the expressions which appear in programs are typically simple

[Knuth71,Tanenbaum78], memory-to-memory operations are desirable [Myers78]. Based on statistics obtained by hand-compiling the mix of statements given in [Bloom74], we estimate that Nebula code may be more compact than code for the PDP-10 or IBM 370 by a factor of 2. Although this statement mix is compiler-oriented, rather than application-oriented, it is likely that application code would show a similar degree of compression.

Nebula is like the VAX architecture in many ways. Thus, it seems reasonable to believe that the two would show comparable code compactness. Nebula has a number of advantages over VAX with respect to JOVIAL. The most important is that data allocation in Nebula has a better correspondence to JOVIAL than does that of VAX. This is a serious problem in VAX for JOVIAL implementations. It is discussed in detail in chapter 4. Another advantage is that the addressing modes in Nebula correspond more closely to those found in JOVIAL programs. This is particularly true for based or parameter tables, where indexing must be applied after indirection. The parameter passing mechanism is slightly more compact in Nebula than in VAX. Also, the fact that sizes are associated with operands, rather than operators allows more efficient code to be generated.

On the whole, Nebula is better suited for supporting JOVIAL programs than are virtually all other current architectures. There are some inefficiencies, but they should not be particularly difficult to remedy.

4. JOVIAL Implementation Issues

For each of the language-related areas below, we discuss problems which arise in implementing JOVIAL for typical targets, and how well Nebula supports those language features. If we feel that better support could have been provided, we analyze the effects of the problem and recommend modifications to Nebula to correct the problem, or sketch a way around the problem. In certain cases the solution to one problem affects the solution of others. Our recommendations for these related problems are collected together in section 7.

4.1 Parameter Labels

JOVIAL allows labels and procedures to be passed as parameters. In both of these cases a reference to the formal parameter is a reference to the actual parameter in the environment in which it was originally passed as a parameter. This implies, for example, that a branch to a parameter label may be used to jump part of the way out of a recursion. For this reason, the stack pointer must be passed, along with the value of the label itself, if the goto is implemented as a branch directly to the label. If a display is used for up-level references, it is necessary either to save a copy of the display when the label is passed as a parameter, or to unwind the stack and restore the display to its

state at the call which passed the label.

An alternative implementation is to pass a flag back to the caller to indicate whether a branch has occurred. Extra code must then be generated in the caller to branch, based on the value of this flag.

We are unaware of any hardware which provides the high level support required by JOVIAL for parameter labels. Some architectures such as CAPS allow branches to parameter labels, complete with stack unwinding, but the branch is always to the occurrence of the label in the most recent invocation of the procedure containing it. This is not necessarily the correct one, according to JOVIAL rules.

One of the most difficult problems in implementing JOVIAL on Nebula concerns the handling of label parameters. These problems arise because the context stack is hidden from programs executing in user mode, and the fundamental differences between the semantics of parameter labels and Nebula exception handling mechanism. The differences are these: exception handlers are statically associated with procedures, but parameter labels are dynamically associated with environments; and, there is one exception handler for a procedure, but a number of label parameters may be passed for one call. When an exception is raised, the most recent exception handler is invoked, but a label parameter may refer to the label in a previous invocation of the

procedure. When a goto to a parameter label is performed, both the context stack and the data stack must be trimmed back to the proper level.

As described above, it is relatively easy to restore the data stack to the proper height, if the stack top at the call is saved with the label parameter. However, the user program has no direct mechanism for setting the context stack back to its height at the call. It is possible to create an exception handler for each procedure which passes a parameter label, to raise a parameter label exception at a goto parameter, and to check the data stack level at the handler, to determine if the exception needs to be reraised to get to the proper stack levels. The only problem here is that the stack height taken from the formal label parameter must be passed back from the point of the goto, back to the exception handler. This could presumably be done by storing the pointer in a global cell.

Although it is possible to implement label parameters on the current Nebula architectures, it would be useful to introduce some additional instructions to make the process simpler, and the code more compact. These instructions are: get context (GETCON) and branch with context (BRCON). Get context would return a context marker into its operand. This marker could be implementation dependent. Using it as an operand in a context other than where a context marker is required would produce unpredictable results.

Branch with context would take two operands, an address operand and a context marker. Executing a BRCON would cause the context stack to be popped to its state when the context marker was created. The next instruction executed would be taken from the location specified by the address operand. Note that a displacement is not sufficient for this instruction, since the label is execution-time variable.

4.2 Parameter passing and referencing

JOVIAL allows the programmer to specify the method by which parameters are passed: by value, by reference, or by result. Most machines do not provide such support with respect to parameter passing, letting the compiler use general purpose primitives for this purpose. Some architectures which support stacks, such as CAPS, allow parameters which have been pushed onto the stack to be referenced as local data within the callee.

The most efficient way of passing parameters is highly dependent on the particular architecture, but some generalizations can be made. The compiler is often required to generate a prologue and/or epilogue in the called routine for the purpose of copying value parameters in, and reference parameters out. Whenever possible copying should be done by the callee, rather than by the caller, because calls outnumber procedures.

Nebula provides some high-level help in passing parameters but not in the full generality required by JOVIAL. The Nebula parameter passing mechanism is equivalent to passing all parameters by reference; value and result parameters are not handled directly. This is not a big problem, however, since value parameters may be moved into local space during the prologue, and result parameters may be moved out during the epilogue. This does imply, though, that return statements in the JOVIAL source be translated into branches to the procedure epilogue, rather than into Nebula return instructions. Generating prologues and epilogues and handling returns in this manner is common and poses no great problems, but it does become more serious with small procedures, however, where prologue/epilogue overhead can become a significant percentage of the entire procedure in both space and time.

(In Ada formal parameters cannot always be moved directly into the actuals by epilogue code, due to constraint checking requirements [Davis81]. It would be possible to pass compiler-generated temporaries for those parameters whose constraints did not match those of the formals, and to copy the returned values from those temps to the actuals in the caller. This would not be a bad strategy if the constraints usually were compatible, but would double the number of moves if the constraints were always more stringent for the actual parameters.)

One problem which is potentially major is in referencing up-level parameters. Since the context stack is protected from user programs, it is not possible to reference up-level parameters in a straightforward manner. It would be possible to pass the up-level parameters on to the inner procedures in which they were referenced, but procedure parameters would complicate this process. Another possibility is to copy the parameters or their addresses into local storage.

The CALL mechanism alone would not be sufficient except when the compiler could be sure that there were no up-level references to parameters; determining this would, of course, require extra analysis by the compiler. Note, however, that in JOVIAL this analysis is always possible; in Ada it is impossible in the presence of separate compilations [Davis81].

If too much additional overhead is incurred due to procedure calls and parameter passing, JOVIAL programmers may be tempted to sacrifice readability for efficiency, by using global variables or by avoiding procedure calls altogether.

Using a bit mask in the procedure header to indicate the parameter passing mechanism should be considered. To be compatible with JOVIAL parameter passing, at least four modes need to be provided: copy in, copy out, copy in and out, and reference. These correspond to BYVAL input, BYRES output, BYVAL output and BYREF parameters.

Recommendations for a mechanism to allow references to up-level parameters are discussed below under "Operand Addressing".

4.3 Data Referencing and Storage Alllocation

JOVIAL allows static or dynamic allocation to be specified for data which is declared in procedures. Thus, there are implicit requirements on the data accessing mechanisms of the target machine.

The requirements with respect to recursivity and reentrancy in JOVIAL are very much like those of Ada with but a few exceptions. The most important have to do with label and procedure parameters, which are discussed in Sections 4.1 and 4.4. The other important difference is that JOVIAL procedures may be recursive, reentrant or neither. This is actually a relaxation of the requirements for individual procedures because it permits implementations which are recursive but not reentrant, or vice-versa. However, the net effect on the architectural requirements is the same.

Recursion is implemented by means of a stack. JOVIAL, like Ada, allows nested procedures and up-level references to data declared in containing scopes. Parameters to enclosing procedures may be referenced, as well. There are several ways in which this is

done. One is through the use of a display. This allows up-level references to be made relatively efficiently, but requires additional work at proc entry or proc entry and exit, depending on how the display is implemented. An alternative is to search through the static links at references. This makes procedure calls cheaper, but references to up-level data more expensive. Hybrid methods have also been used [Hawkins63].

On most machines there is little direct support for recursion. Typically, scalars are referenced by using indexed instructions, and the local frame pointer must be explicitly added to the subscript for referencing tables. (The IBM 370, which allows an index and a base register in the same instruction is a notable exception.) Some stack-oriented architectures such as the CAPS and the Pascal Microengine do provide good support for recursion. The Motorola MC68000, which is a more conventional architecture, has instructions (LINK and UNLK) which assist in allocating and deallocating local data.

Reentrancy presents some different problems. Since reentrant procedures may be executed concurrently by different tasks, it is necessary that data accesses refer to different storage locations for different threads of control. There are different levels at which programs may be reentrant. Any routine may be sharable on hardware such as the DEC-10, which has code and data base registers which are hidden from the user. If the system uses the same code base but a different data base for different tasks,

reentrancy is achieved. A similar effect can be achieved by altering the page map on systems which provide virtual memory. If such hardware support is unavailable, reentrancy can be achieved making all data references relative to that address. This approach, like the one discussed for recursive procedures, requires, on most machines, extra computations to reference table entries.

Nebula does provide some support for recursion, in the form of the stack pointer register and the stack oriented instructions (push and pop). The addressing modes are certainly adequate for supporting recursion, although having an address mode which assumed the existence of a local frame pointer could save some space. (It would then be possible to access a certain portion of the local frame with a one byte operand specifier.)

Up-level references are not very well supported, for several reasons. There is no direct support for a display or for following the links. In addition, because only the stack pointer is inherited from the caller, it would be more difficult to keep the display or a pointer to the display in registers.

4.4 Parameter Procedures

In the general case, parameter procedures present potentially greater problems than do parameter labels. The environment must be recorded, in order to implement calls to parameter procedures in accordance with JOVIAL since they (parameter procs) must be executed in the environment in which they were passed. The actual method used to achieve this varies according to the method used for up-level referencing, but one common way to implement it is to copy the display at the point at which the procedure parameter is passed, and to pass the copied display along with the parameter. Another possibility, if up-level references are made using the static links, is to pass a pointer to the current stack frame along with the procedure parameter, and to mark the stack so that up-level references from within the parameter procedure will bypass those stack frames between the parameter proc's stack frame and the one in which it was created.

Nebula provides no special support for the implementation of parameter procedures. Our main concern is that if the architecture is modified to provide support for up-level references for Ada, that the changes may be incompatible with JOVIAL, because Ada does not permit procedures to be passed as parameters. If a local storage allocation mechanism is added to Nebula, and the methods of data accessing are hidden from the user program (in a manner analogous to that currently used for parameter references), it may be impossible to implement

parameter procedures, unless special facilities are designed in to the architecture.

4.5 Character string operations

JOVIAL allows moves to and comparisons of character string operands which are not of equal length. A further complication is that the byte function allows strings, whose lengths are not known at compile time, to be manipulated. Although many machines support character moves for equal length operands, there are few (VAX is a notable exception) which provide an efficient way of implementing moves for operands of unequal length whose lengths are not known at compile time. An inline implementation of such a move would require taking the minimum of the two lengths, performing the move and then filling with blanks if the receiver is longer than the source. Alternatively, if the operands do not overlap, the destination could be filled with blanks first, and the source value moved in later. Assignment is further complicated by the rules for overlapped move (see below). Unless the hardware gives the correct results for an overlapped move or unless it can be determined that fields do not overlap, the compiler must generate an extra move to a temp, control the order in which bytes are moved, or generate a subroutine call. This can require extensive analysis, if worst-case code is to be avoided.

Nebula gives good support in the form of the move block (MOVBK) instruction, for moves in which the source and sink are the same length. MOVBK together with move multiple MOVM provide fairly good support for moves of constant length where padding is required. However, the support for moves where the source and sink may differ in length, or one is of variable length, is only moderate. The remarks made above with respect to variable length moves apply to Nebula. An added complication is the fact that move multiple takes an unsigned count. This means that the fill count must be computed as MAX(length(sink)-length(source),0) or that the MOVM be skipped if the fill count is negative.

Since it is difficult to generate efficient code for character moves and compares which are potentially overlapped and which involve variable length operands, we recommend that instructions be added to Nebula to facilitate these operations. These instructions should allow, at the very least, lengths and addresses to be specified for the two operands. Additionally, it is desirable to allow a first byte to be specified. Although the first byte can be subsumed into the address calculation, specifying it separately corresponds more closely to the JOVIAL language usage, and allows more efficient coding of accesses to subscripted character strings. JOVIAL only provides blank filling, but it may be desirable to allow a fill character to be specified, if such filling is supported at the source level by other languages for which Nebula is to be used.

Move characters (MOVCHS) should perform truncation or filling with blanks on the right, as appropriate. The instruction should provide for overlapped source and sink, and give the same results, whether or not the two fields are overlapped. Since the instruction execution time is potentially long, the instruction should be interruptable.

Compare characters (COMCHS) should perform a comparison between two character strings, as if the shortened had been blank padded on the right to the length of the longer. Overlapping fields are not an issue here.

Overlapped Move

The semantics of JOVIAL assignment require that the fact that the source and the sink overlap not affect the assignment. This means that unless the target computer performs overlapped moves in a manner which is consistent with JOVIAL, or unless the compiler can determine that overlap is impossible, it must generate extra code. This code may take either of two forms:

1. The compiler may generate code to select the direction of the move, or more simply,
2. The compiler may move the sink to a temporary, move the source to the sink, and then move the temporary to the source.

Alternatively, a subroutine call could be generated.

Since move block (MOVBK) is defined so that overlapping operands do not affect the result, overlapped moves are not a problem in Nebula, provided that the operands are the same length, or are of constant length. Recommendations for handling overlapped moves are discussed above under "Character String Operations".

4.6 Bit strings

The maximum length of a JOVIAL bit string is MAXBITS, an implementation parameter. Because MAXBITS determines the maximum allowable size for table entries, any reasonable JOVIAL implementation will choose MAXBITS to be significantly greater than the number of bits in a word. This causes problems on many machines because normally operations on bits are only supported in the hardware for word-size or smaller operands. Although some machines, such as the VAX, allow bit addresses to span word boundaries, they normally do not support bit operations on bit strings which are longer than a word.

A similar problem is presented by the bit function when it is applied to large operands, unless it is known at compile time that the number of bits is small. Code can usually be generated in line for applications of the BIT function to operands which

are known to be contained in a word. The code sequences generated for the BIT functions with a constant first bit and number of bits are identical to those generated for part-word operands. The sequences for variable first bits and number of bits typically involve shifts or special extract or deposit instructions if they are present in the hardware.

Nebula provides good support for BIT functions applied to word size or smaller operands, or BIT functions whose first bit and number of bits are such that the results are guaranteed to be byte aligned and an integral number of bytes long. Support for longer bit strings is only marginal. Although it would be possible to generate code to perform long variable bit moves inline, it most likely would not be done in practice, because the code sequences would be rather lengthy. It is likely that a subroutine call would be generated. This would mean that there would be a threshold size for bit strings, above which they were more inefficient. Although it would be convenient for the compiler to be able to generate code directly for the various operators which can be applied to long bit strings, the payoff in terms of efficiency in actual practice would be quite small, except in those rare programs which made heavy use of those sorts of operations. Rather than proposing that all instructions be applicable to arbitrary bit operands, we suggest a more modest addition to Nebula, namely long bit move and compare instructions. The operation described here satisfies JOVIAL semantics; support for other languages may require a

generalization.

The proposed instruction "Move Bits" (MOVBTS) moves the contents of one bit field to another, padding on the left with binary zeros, or truncating on the left as appropriate. There are six operands to this instruction; for each of the source and sink, a pos, size and base are specified. These are calculated as in the bit field instructions, but the size restriction is removed. Since this means that the instruction potentially takes a long time to execute, it must be interruptable.

Another proposed instruction "Compare Bits" (CMPBTS) compares two bit strings. The shorter is padded on the left with zeros. Although JOVIAL does not permit bit strings to be compared for ordering (only equal and not equal are permitted), it would make sense to define the comparison so that it sets the N flag bit if the first operand is logically less than the second. This instruction would have the same operands as MOVBTS.

4.7 Truncation and Rounding

JOVIAL contains requirements for truncation and rounding which are not often supported in hardware. Users are allowed to specify rounding or truncation towards zero or negative infinity. The specifications may be associated with specific

items of any numeric type (integer, fixed point or floating point). Thus, all three types of conversions may appear in a single program.

Most hardware does not support the three types of length reduction called for by JOVIAL. Items which use the type implemented in the hardware are handled efficiently. Subroutine calls must often be generated for the operations which are not directly supported.

Nebula provides good support with respect to the types of rounding supplied; the three mentioned above are handled and there is round toward positive infinity, in addition. However, the type of rounding is determined by the flag bits in the Auxiliary Status Register (ASR), and is not associated with the result field as is required by the language definition. The desired results could be obtained, however, by altering the ASR prior to storing the result. This would increase the size of the code generated, though, and would tend to discourage heavy use of rounding and truncation.

We have several recommendations regarding ways to make Nebula truncation and rounding more usable for implementing JOVIAL. First, the fixed point operations (MULFIX, DIVFIX and SCALE) should round or truncate in a manner consistent with the floating point operations. (This is to say, if the ASR is used to indicate whether rounding or truncation is to occur, then it

should also be used for fixed point. However, it may be advisable to have a separate set of bits to control fixed and floating rounding and truncation. We will suggest alternative methods below.) JOVIAL requires truncation toward zero and negative infinity as well as rounding. Since truncation toward positive infinity is included in Nebula for floating point numbers, it should be allowed for fixed point, also.

Another recommendation is that there be a convenient way to locally change the rounding. We have two suggestions as to how this could be done; which one should be implemented depends largely on the availability of op codes. From the JOVIAL point of view, it is desirable to be able to specify the rounding and truncation for each operation. For example, rather than just having a move floating instruction, there would be move floating and round, move floating and truncate to zero, etc. This would have to be done for all operations which required rounding or truncation. The disadvantage of this approach is that the number of such op codes is quadrupled (although JOVIAL would permit rounding to positive infinity to be omitted).

Another approach which is not as efficient or esthetically pleasing, but which is more conservative of op codes, is to allow a flag which specifies the type of rounding, preceding operations which perform rounding or truncation.

It would also be possible to have specific rounding operations

which followed arithmetic operations, but this is less satisfactory since precision may already have been lost.

4.8 Operand Sizing

JOVIAL programmers are allowed to specify sizes for items. This implies that items in densely packed and specified tables must be extracted before they can be used in computations. On most machines support for such operations is limited, consisting of shift and logical operations. The problems associated with packed and specified table items are discussed under "Part-word operands".

On byte-oriented machines it is common even for scalars to require sizing if they are allocated to storage units smaller than a word. The fact the JOVIAL associates sizes with data, but machines generally associate them with operators, causes extra code to be generated to convert all of the operands of a given operator to a common length.

Nebula is much better than most machines with regard to allowing operations to be performed on different sized operands, because sizes are associated with operands, rather than with operators. In the design, however there is one potential pitfall, which applies to languages such as Ada as well as to JOVIAL. Consider

the following statements:

```
S16 = U8 + S16;
S32 = U8 + S16;
```

Where S and U indicate the signedness of the operands and the integers give their sizes. If the first addition is performed using signed addition (ADD) and the value of U8 is greater than 127, U8 will be sign extended and the answer will be incorrect. If unsigned addition (ADDU) is used, the correct answer, is obtained for the first statement, but not the second if S16 is negative, since S16 will not be sign extended.

One solution to this problem is to allocate two bytes for items which are declared U 8, but this is wasteful of space. The "correct" solution would be to associate the sign with the operand rather than the operator, but this would pose coding problems for the operand specifiers.

4.9 Part-word operands

JOVIAL permits the user to specify part-word operands in dense-packed and specified tables. These operands may be used in any operations which are appropriate to their data type. Support for such operands is typically quite limited. In very few cases

is it possible to use such operands directly. (Compare field (CMP) on the VAX is one example.) It is almost always necessary to extract operands first. In addition, there is usually not very much hardware support with respect to extraction, which normally must be performed using various combinations of shifting and masking instructions. Some machines, such as the VAX, do have extract field instructions.

Nebula is better than most machines in this regard, although part-word operands must still be treated specially. Support for single bit operands is relatively good with clear, set, invert, and test operations available, but there is no specific operation for setting a single bit to a variable value. (Store bit field (SBF) can be used, but it could have been used for clearing and setting bits also.)

Support for operands whose size is bigger than a bit but smaller than a word consists of the store bit field (SBF) and the load bit field (LBF and LBFS) operators, and numerous instructions which handle byte and two-byte operands.

Code for part-word operations on Nebula would be somewhat less efficient than it could be, due to the fact that it is necessary to load, compute and store rather than to perform the operation directly to the target. Our recommendations for improvements in this regard are discussed below under "Operand Addressing".

4.10 Optimization

Although optimization is not required by the language specification per se (with such exceptions as short circuit evaluation of logical operators), the nature of the tasks for which JOVIAL is used require code which is economical with respect to both time and space.

The problems of generating good code are compounded by poor machine support for high level language constructs, since special cases require additional analysis by the optimizer or the code generator.

Since Nebula is a relatively regular architecture there are fewer special cases than for most machines. There are a number of machine features which do require extra analysis, if the optimizer is to take advantage of them. The two most important are the three-operand operators and the large number of addressing modes.

Three-operand operators make it easy to generate code for statements like:

```
A = B op C;
```

in the absence of common subexpressions. However, when common subexpressions are to be considered, the optimizer must decide

whether it is better to compute the value in a register (or temp) or to compute it directly into A. On a machine where all computations must be done in registers, this is not an issue.

The large number of addressing modes in Nebula provide additional opportunities for optimization. Since it is possible to save three bytes each time a byte offset is used in place of a word offset, it behooves the optimizer, not just to use short addresses where they happen to occur, but also to create additional opportunities by loading base registers.

Again, this is a feature which can be ignored by any compiler which chooses to do so, but which can allow a considerable saving to be made in terms of program size.

4.11 General Machine Idiosyncracies

Many instruction sets, particularly those for older machines were apparently designed with insufficient regard for generating code for high level languages including JOVIAL. The architectures contain numerous examples of instructions which do not correspond well to operations in high level languages.

Machine idiosyncracies take a variety of forms. One of the most common, and one which complicates code generation considerably is

inconsistent use of registers. Most machines with index registers do not allow register zero to be used for indexing. Other machines have requirements for the use of double registers or register pairs for certain instructions, typically multiplication and division. Other machines require that specific registers be loaded prior to the execution of certain instructions. These peculiarities make it more difficult to generate good code.

Another common irregularity is the restriction of addressing modes for certain types of instructions. Examples of this are Move character (MVC) on the IBM 370 (only a single register may be used) and the double precision instructions on the MIL-STD-1750A (only register and direct addressing are allowed). For a more complete discussion of this issue see [Wulf81].

Nebula is less idiosyncratic than most architectures, but is not perfect in this regard. Some operators such as XOR, MOD and REM appear only in the three address form while other similar operators such as AND, OR and DIV appear in two and three operand forms. This requires that these operators be treated specially by the compiler, but does not have a noticable effect on the JOVIAL programmer. We recommend that, unless there is a severe shortage of op codes (which doesn't appear to be the case with less than 128 presently assigned) that both the two and three operand variations be allowed for XOR, MOD, and REM.

Increment and decrement are not symmetrical. It is possible to increment by 1, 2, 4 or 8 but only to decrement by 1.

Procedure descriptions are required to be word aligned. This causes extra work for the compiler and linker.

The effect of moving 64-bit items to registers is not clearly specified in the Nebula definition. Section 5.4 states that "register operands shall be word (32 bit) size". This would seem to indicate that 64-bit operands either cannot be moved to registers, are truncated to 32 bits, or require two registers. Any of these alternatives would require special handling for such operands; this would complicate the compiler. We recommend defining the register set so that, as a minimum, 64-bit floating point operands can be loaded.

4.12 Relationals in value contexts

Relational operators may appear in boolean expressions in either flow of control or value context (i.e., when a relational operator appears in an assignment statement). Normally, there is good support for the flow of control case and it is possible to generate efficient code for IF statements. However, most machines do not provide good support if the value of the boolean needs to be generated, as in a boolean assignment.

JOVIAL introduces an additional complication in that it requires that a single bit logical expression be computed using short circuit evaluation. The way this is achieved for most architectures is to translate a statement like:

```
A = B = C or D = E;
```

as if it had been written:

```
T = 1B'1';
if B = C; goto 1;
if D=E; goto 1;
t = 1B '0';
1: A = T;
```

Nebula offers a means for converting condition codes to boolean values. This makes it simpler to generate code for single boolean assignments such as:

```
A = B = C;
```

However, not much help is provided for the more complicated cases like the one shown above.

These difficulties are more the result of the language definition, than of Nebula. For this reason we recommend no change with respect to the short circuit evaluation.

4.13 Case

Many architectures furnish no special support for the case statement. This is not a big problem for integer selection, because ordinary conditional and indexed branches are sufficient to implement case statements without an unreasonable loss of efficiency. JOVIAL also allows character and bit type selectors. These present more of a problem, since the range of possible values may be too great to allow the use of an indexed jump.

Nebula's case operator provides good support for CASE's with integer and status selectors.

4.14 Loops

Although a number of machines provide some degree of support for loops, JOVIAL loop statements often do not map easily onto the instructions which are intended to provide this support.

One common problem is presented by "increment (decrement) and branch" type instructions. Normally, the decision on whether to branch is made based on a comparison of the counter with zero. Unless the termination condition can be mapped into a test against zero, these special instructions are not usable.

Another problem is that loop instructions are often applicable only to testing at the bottom of a loop, (ala FORTRAN), since they are of the form "increment (decrement) and branch". JOVIAL semantics require that the condition be tested before the loop is executed for the first time. The compiler can move the test to the bottom of the loop if the loop is known to be executed at least once (as indicated by constant bounds), but if the loop bounds are variable, either an extra test must be generated, or else the increment must be subtracted from the initial value so that when the special instruction adds the increment, the correct initial value will be checked against the limit.

JOVIAL presents several additional complications. One is that there are a number of rather general forms of loops. The WHILE loop is more or less standard and presents no particular problems as far as implementation is concerned. The FOR loop which uses a counter (BY phrase in JOVIAL) has a more general termination condition than is normally encountered. Any boolean formula may be used as the termination condition; the condition is not limited to the relation between the loop variable and some numeric value. Thus, even those machine instructions which allow the loop variable to be tested against an arbitrary value do not support JOVIAL loops in their full generality. JOVIAL has another form of FOR loop which is relatively uncommon. The THEN clause specifies an expression which is to be reevaluated and assigned to the loop variable at the end of each iteration through the loop. We are unaware of any loop instructions which

support this construct, and it is unlikely that special instructions would be particularly helpful.

Another minor complication is that JOVIAL defines the value of the loop variable outside the loop in cases where the loop variable is declared explicitly. This means that machines which suppress the increment at loop termination will not conform to JOVIAL loop semantics unless additional code is generated by the compiler.

Nebula provides somewhat better support for loops than most architectures do, chiefly because the increment (decrement) and branch and the loop instruction allow limits to be specified. These instructions do, however, give an incorrect (according to JOVIAL semantics) value for the loop variable upon exit from the loop. This would require either an extra instruction to add the increment once more, or analysis by the compiler to ascertain whether the loop variable's value was required subsequently. The remarks about checking for loop termination at the top and bottom of the loop apply to Nebula, also.

We recommend that the loop instructions be modified so that the loop counter is incremented, regardless of the outcome of the test against the limit.

4.15 Tight tables

Tight tables (those with multiple entries per word) are usually poorly supported by target hardware. On a typical machine, it is necessary to perform a division of the subscript by the number of entries per word in order to obtain the number of the word which contains the entry and the number of the entry within that word. Then, the appropriate entry must be extracted from the word using one of several possible extraction sequences (see part word data).

Nebula provides good support (load bit field instructions LBF and LBFS) for the extraction itself, but less support for the address calculation. In fact, the support for the address calculation itself is slightly worse than that found on most machines, because division and remainder are separate instructions, whereas it is common for integer division to produce both a quotient and a remainder.

It is unfortunate that the language requires an integral number of entries per word, since it is easier to generate code for Nebula when the word boundaries are ignored. In such a case, the load bit field operations would be sufficient.

Nebula does provide good support for tight tables when an integral number of entries fit exactly in a word, because the load bit field operations allow full sized integers for the "pos"

operand. In this case, the pos may be computed as

entry-number * bits-per-entry

and the extract done directly.

For this reason, it may be to the JOVIAL programmer's advantage to define tight tables so that an integral number of entries fill a word exactly.

4.16 Star tables

Formal parameter tables with variable bounds (star tables) are not often well-supported by hardware. These tables pose problems for several reasons. First, address computation is more difficult because the lower bounds must be subtracted from the subscripts at execution time. Also, star tables which are passed BYVAL or BYRES require the allocation of an object whose size is not known until execution time.

4.17 Fixed Point Arithmetic

Fixed point arithmetic often presents problems due to inadequate hardware support. Difficulties arise both because there is no direct support for fixed point and because the primitives

provided do not match language requirements. Two of the most common problems are loss of precision and scaling. Precision may be lost if ordinary integer arithmetic is used for fixed point operations. This is particularly true for division, because integer divide instructions often produce just a single precision quotient. Scaling is often inefficient because "arithmetic" shifts often do not produce arithmetic results. On two's complement machines right shift is not equivalent to division by a power of two for negative numbers, and on one's complement machines left "arithmetic" shift is not equivalent to multiplication by a power of two.

The definition of MULFIX and DIVFIX are somewhat imprecise in that no indication is given as to whether bits may be lost in the shifting process. If it is possible for bits of the product or dividend to be lost due to shifting and before the final result is stored, these instructions may not be usable in all cases. If the instructions are equivalent to infinite precision operations with truncation at the storing of the final result, Nebula provides good support for fixed point arithmetic operations. We assume that this is the intention of Nebula, since Ada has essentially the same requirements. Integer addition and subtraction and fixed point multiplication and division can be used for the four basic operations.

The one area in which Nebula is weak is in regard to truncation and rounding. Since integer and floating point operands are also

affected, truncation and rounding are discussed in a separate section.

4.18 Data Allocation

JOVIAL is very much a word-oriented language. Tables may be defined in terms of words per entry, items in specified tables start at a given bit within a word, and tight tables are allocated with an integral number of entries per word.

Because JOVIAL is word-oriented, there are sometimes problems mapping language constructs onto a machine architecture. One common problem arises when machines start the low order byte of integers in the byte with the lowest address. In such cases, characters run from the most significant character to the least significant, in order of increasing address, but integers go from least significance to greatest. This is contrary to the assumptions which JOVIAL makes about the ordering of bits and characters within a word. This can affect the efficiency of accessing bit strings; for character strings it may be necessary to represent values in machine dependent format, which may cause transported programs to fail.

One machine for which this is the case is the VAX. It provides instructions for handling integers, characters and bits, but in

order for an implementation to correspond to the rules of JOVIAL, it cannot take advantage of the hardware to its fullest extent. One would like to be able to use the character move instructions for character strings; that implies using the hardware character allocation scheme. One would also like to be able to use the hardware to perform arithmetic operations; that determines the allocation for integers. Using the hardware allocation for bits, though, causes problems because bit 0 of an integer is the least significant in the machine but JOVIAL defines it to be the most significant bit. This problem could be remedied if an actual conversion was done for converting integers to bits and vice versa. (Normally no code is generated for such a conversion.) Another alternative is to allocate data as specified by JOVIAL, and not make good use of the hardware.

Fortunately, although Nebula resembles VAX in a number of respects, bits, characters and numbers are all allocated so that the most significant byte is the first byte (i.e., the byte with the lowest address). Thus, this is not a problem, given the current definition of Nebula. We mention it here to caution against any changes in the definition which would introduce this problem.

4.19 Abort statement

The ABORT statement is unsupported by current architectures. When an ABORT statement is executed, control is transferred to the label specified in the ABORT clause of the innermost call. Normally this is handled by calling a library subroutine which unwinds the call chain and pops the stack back to the appropriate levels. If a display is used for outer scope references, this routine restores it to its condition when the abort clause was encountered.

Since the context stack is not directly accessible to user programs, this approach cannot be used if the CALL instruction is used for subroutine calls. Fortunately, there is a better correspondence between the ABORT statement and Nebula exception handling than there is between gotos and the exception handling. When an ABORT statement is executed, the abort "handler" is the label which was specified in the most recently executed call which contained an ABORT clause. This corresponds rather well to the method for handling exceptions; the most recent handler is invoked. The only differences is that exceptions are associated with procedures and ABORTs with calls, but since there can be only one active call from a procedure at a given time, this is not a problem. The sequence:

```
EXCEPT   abort-label   ;   specify label
CALL     proc
```

```
EXCEPT   #0              ;   disarm abort
```

can be used satisfactorily. Although it would be possible to define an ABORT label operation which would eliminate the need to turn off the exception handler after the return from the call, this would be a gimmick. The savings in terms of space would be relatively small - only two bytes per clause. We do recommend, however, that an ABORT exception code be reserved, to avoid a conflict in the event that JOVIAL and Ada programs are linked together.

5. Other Issues

There are certain other issues which do not cause problems in implementing the MIL-STD-1589B version of JOVIAL, but which do present difficulties in implementing other languages or other versions of JOVIAL. They are discussed here to allay any fears that the reader may have with regard to these particular topics.

5.1 Parallel Tables

One area of JOVIAL J73/I which caused a number of implementation problems on a variety of machines was parallel tables. It was then possible to have multiple-word items in parallel tables. This meant that the words of such an item were allocated to discontiguous storage locations, requiring extra code to be generated whenever such an item was referenced.

The current definition of JOVIAL (MIL-STD-1589B) has eliminated these problems, because it restricts items in parallel tables to being word-size or smaller. Thus, parallel tables cause no particular problems.

5.2 Checking

One of the concerns about the Nebula architecture is whether it provides sufficient support for checking, or whether the three operand operators are rendered useless by range checking requirements. JOVIAL, unlike Ada, has virtually no formal requirements for checking the legality of subscripts or for performing range checking for assignments. In fact, there are no explicit requirements for checking.

Some checking is traditionally done, however. Reasonable implementations check whether the stack will overflow when space is allocated. Exponentiation library subroutines normally check for errors such as raising a negative number to a non-integer power, or raising zero to a negative power.

It should be kept in mind that some features of Nebula are more useful in JOVIAL, where checking may be ignored than in Ada which required checking. (For a discussion of the problems presented by Ada in this regard see [Davis81].) Three operand operators, the general indexed modes, and the parameter passing mechanism are all more usable in the absence of checking, than in its presence. They are quite useful for JOVIAL, although parameter passing could be made more so.

5.3 Tasking, Input/Output

There are a number of issues such as input/output and tasking which are important to the implementation of embedded systems, but which are outside the scope of the JOVIAL language definition. In actual JOVIAL programs, input/output and tasking are done through procedure calls to assembly language routines. There is no standard in the language as to what routines exist, and we know of no de facto standards. The only formal requirement in these areas is that it be possible to support reentrant procedures. This is discussed under "Procedures".

Because JOVIAL has no explicit requirements for tasking or input/output, we did not investigate these areas in detail. An architecture which can support Ada should provide sufficient support in this regard.

5.4 Registers

Nebula's register handling is unusual, but not unique (the TI990 also gives each procedure its own set of registers). The fact that each procedure "owns" its registers has advantages and disadvantages with respect to efficiency, but we feel that it is primarily an advantage. The disadvantages are that it is impossible to pass parameters or to maintain global information

(such as a display) in the registers and that for certain implementations there may be a speed penalty. The advantage is that it is not necessary to save the registers at procedure calls.

6. Effect of Nebula's Shortcomings on JOVIAL Programmers

Most of the problems which JOVIAL presents to JOVIAL implementations should have little effect on the JOVIAL programmer. The only JOVIAL feature which would have an unreasonable implementation on the 1862A version of Nebula is parameter labels. These are likely to be used infrequently, anyway, and are likely to be used only when there is no reasonable alternative. For that reason we do not expect that Nebula will have much impact on their use.

The fact that parameters are passed by reference in Nebula will have the effect of discouraging the use of small procedures, due to the fact that prologues and epilogues will be a significant fraction of the total size of those procedures. However, this is no more true on Nebula than on ordinary register-oriented architectures.

Inefficiencies in handling long bit strings and mixed rounding and truncation may discourage some efficiency-minded programmers from using these language features. Explicit truncation and rounding are likely to be used where they are necessary and cannot be avoided, so the actual effect should be small.

7. Additional Recommendations

This section contains recommendations for modifications to Nebula which are related to several of the problem areas discussed above, as well as suggestions for improvements which do not really address problems, but which could contribute to the efficiency of Nebula for JOVIAL programs.

7.1 Procedure Calls and Space Management

We feel that the hardware could be of greater assistance in implementing procedure calls efficiently, particularly with regard to the bookkeeping which must be done at proc entry and exit. This could be done with generated code, but providing help using hardware would guarantee a uniform convention across compilers for different languages, would enable more efficient references to local data, since assumption could then be made about the location of the local frame pointer, and could provide an additional degree of protection by preventing a reference to one stack frame from accessing data from another.

Our proposal involves a modification of two instructions, CALL and RET, and addition of another, SPACE. In addition, the procedure description should be modified to contain an indication of how much local storage to reserve. CALL should be altered to

set up the local frame pointer (possibly register 2) based on the stack pointer, and to bump the stack pointer by the amount of storage specified in the procedure description. Before the stack pointer is modified, a check should be made to insure that the stack limit is not exceeded. The value for the limit could be kept in a register in I/O space.

A RET would undo the allocation performed by the call. If the stack pointer and local frame pointer were both kept in registers, this would be automatic, since the old values of those registers would be restored from the context stack.

It would also be useful to allow space to be obtained without executing a CALL. This is necessary in JOVIAL for obtaining space for star tables which are passed by value, and in other languages, such as Ada, for arrays with dynamic bounds. The SPACE instruction would have a single explicit argument, the amount of space to be obtained. The stack pointer, the frame pointer, and the stack limit would all be implicit.

One aspect of the hardware design which must be addressed by the hardware designers, but which does not necessarily have to be part of the standard, is the mechanism for up-level references. Either a display-oriented or static-link-oriented scheme could be used. Provided that facilities are provided in the hardware for up-level references, the compiler need not know which method was chosen. It is necessary, however, to be able to handle both

parameter labels and parameter procedures if JOVIAL is to be implemented on such an architecture. Our recommendations for parameter labels were mentioned above. For parameter procedures a "call and set context" instruction would be necessary. This instruction would either set up the appropriate static links or alter the display, depending on the method of implementation which was chosen.

7.2 Operand Addressing

The most important factor in achieving code compactness is conciseness in referencing operands. This is particularly true in an ISA like Nebula which provides such a large virtual address space. As mentioned above, those architectures which implement JOVIAL most efficiently are the ones which handle operand addressing in the most efficient manner. Operand addressing in JOVIAL is more general than even Nebula, which provides more addressing modes than do most machines. Examples of addressing types which are not handled directly in Nebula are subscripted subscripts and pointers to pointers to pointers. Ideally, the compound addressing modes in Nebula would allow compound modes to be their operands, but this would prevent the address calculation mechanism from being a finite state machine. For this reason we are not suggesting that this generalization actually be added to Nebula.

There are, however, several opportunities to make operand addressing more efficient: local data, part-word items and items in tables.

7.2.1 Local Data

It is already possible to address those local data items which are allocated to registers with single-byte operands. It would be advantageous to be able to address memory operands with a single byte also in those procedures whose local data did not fit in the registers. A possible approach here is to introduce an operand specifier which assumes that the data is based on the local frame pointer; this would require a convention for the location of the local frame pointer.

7.2.2 Up-Level (Dynamic) Data

If the architecture provides support for allocating data on the runtime stack, it should also provide an operand specifier type which allows up-level references. This could be a compound mode which would take a (static) nesting level number and an operand specifier. Ideally, the present compound modes (scaled index, unscaled index and extended parameter) should be allowed as

operands of this new mode. The data accessed would be that which was at the offset specified by the operand from the local frame pointer for the procedure whose nesting level was specified.

7.2.3 Up-Level Parameters

If the Nebula parameter passing mechanism is to be used efficiently, there should be a means by which up-level parameters can be referenced. This can be accomplished in several ways. One is to add a mode which would allow references to parameters at the nth static level. Another is to move the parameters' values or addresses into the local stack frame when the call is executed. It would then be possible to access the parameters through the same mechanism which would be used to reference up-level data. (This mechanism needs to be added also. See "Up-Level (Dynamic) Data" above.)

7.2.4 Part-Word Operands

Because part-word operands are common in JOVIAL, and can be used in a number of different contexts, we feel that it would be advantageous to have an accessing mode which defined a part-word field. There are two possible approaches. One is to let the

first bit and number of bits be separate operands in a super operand mode. This mode would have a total of three operand specifiers: the first bit, the number of bits and the address. The field defined by the part-word operand mode would be the same as the field specified by the corresponding three operand descriptions in the bit field intructions (LBF, LBFS, and STF). The correspondences between the JOVIAL and Nebula nomenclatures are as follows: first bit = pos, number of bits = size, address = base. Ideally, there would also be an indication as to whether the operand was signed. This could be done by introducing signed and unsigned part-word operand specifiers. The operand would then be extended to the appropriate size by sign-filling or zero-filling, depending on the operand specifier.

An alternative which is less general, but more space efficient, would be to use an operand descriptor which would contain the first bit, number of bits, an indication of signedness and a base address or offset. This is reminiscent of the scheme used on the DEC-10 but with several important differences. The most important is that the index or base registers, if any, are specified in the operand, rather than in the descriptor. The compound addressing mode in this case would contain two operand specifiers, one for the descriptor, and one for the base. The address obtained from the base would be added to the offset in the descriptor to form the actual base address, which would then be used with the first bit and number of bits to determine the location of the field.

7.2.5 Table Data

Nebula provides a special addressing mode (scaled index) for accessing data which is allocated in 1, 2, 4 and 8 byte units. Many programming languages, including JOVIAL allow arrays of records (in JOVIAL they are called tables). First, table entries are often not 1, 2, 4 or 8 bytes long. Second, if there are multiple items in an entry, the entry size is not equal to the item size. In either of these cases, scaled index mode is not applicable. It would be very useful to have an operand specifier which allowed an arbitrary multiplier. For the common cases it would be sufficient to permit a constant multiplier, but for consistency with the rest of Nebula it is probably a good idea to allow the multiplier to be a regular operand. This compound addressing mode would take three simple operands as arguments: the multiplier, the index and the base. The index and the base would satisfy the same requirements as the same operands in scaled and unscaled index modes. The multiplier would be a value operand. Addresses would be calculated as:

multiplier*index-value + base-address.

7.3 Miscellaneous Operations

There are several operations in JOVIAL which are not supported directly. These include ** (exponentiation), the EQV (bit-by-bit

equivalence) operator and SGN. Implementing exponentiation in the hardware would provide a significant saving (hundreds of bytes) over a software implementation, and, although exponentiation is not as common as addition or multiplication, almost all embedded applications make some use of it. The latter two are not commonly used operators, but they are easier to implement in microcode than they are to generate code for.

In the absence of an EQV instruction, the code generated is usually:

```
XOR  a,b,temp
NOT  temp.
```

SGN is more difficult to generate code for since it involves branching as in:

```
MOV   #1,t
TEST  a
BGTR  1
MOV   #0,t
BEQL  1
MOV   #-1,t
1:
```

We recommend that these operations be added to Nebula. EQV should be defined in the same manner as XOR. There should be

SIGN instructions which take a fixed- (including integer) and floating-point operand and return an integer result: -1 if the operand is negative, 0 if it is zero, and +1 if it is positive.

Other instructions worthy of consideration are MIN and MAX, although there are no such operators in the language.

7.4 Debugging

Debugging is actually an issue which concerns not just JOVIAL, but programming in general. Nebula provides some debugging features; we feel that there are some others which would be relatively easy to implement in hardware, but could be quite difficult to implement in software.

One of the more useful facilities which could be added is a "break on reference" capability. This would generate a break whenever a given location was referenced. This would facilitate tracing as well as the detection of wild stores. In addition, it would make it possible to set breakpoints even for programs in ROM without having to break at every instruction or call. A refinement to this facility would allow the breaks to be generated for all references or only for those which cause the location to be modified.

8. Summary

The Nebula ISA, as described in MIL-STD-1862A, provides better support for JOVIAL programs than do virtually all other current architectures. Some architectures, such as the CAPS, provide a more compact means of expressing those programs.

Although the architecture is basically friendly with respect to JOVIAL, there remain some areas in which it would be possible to make improvements to achieve greater efficiency. We have made a number of suggestions for modifications to the architecture. In relative order of importance they are:

- Additional support for parameter labels.
- Modifications to operand specifiers to more easily accomodate:
  - Local data references
  - Up-level data references
  - Up-level parameter references
  - Table data
  - Part-word data
- Additional support for parameter passing
- Additional support for storage allocation. (May also require support for parameter procedures)

- Support for truncation,filling and overlapped operands in character moves.

- Support for truncation and rounding of arithmetic operands on a local basis.

- Addition of miscellaneous JOVIAL operations: exponentiation (**), equivalence (EQV), and sign (SGN).

- Addition of two operand forms for operators XOR, MOD and REM.

- Definition of loop operators to conform to JOVIAL semantics (so loop counter value exceeds limit upon exit from the loop)

- Additional debugging support

- Addition of miscellaneous operations: MAX, MIN.

We also wish to call attention to problems which could arise if Nebula is changed without regard to JOVIAL.

- Bits, characters and numbers should all be allocated in the same direction (from the most- to least-significant bit). Otherwise, it will be difficult to implement one or more of these types efficiently and still conform to the rules of JOVIAL.

- A mechanism for allocating local data must take into account the fact that JOVIAL permits parameter labels and procedures. In particular, there must be a way to save environments.

9. References

Arnold, R., "The Nebula Architecture: Ada Issues," Ada Letters, May-June 1981, pp. 11-17.

Bloom, B., et al, Criteria for Evaluating the Performance of Compilers, RADC Technical Report RADC-TR-74-259, 1974.

Davis, M. and Stryker, D., Nebula as a Target for Ada, Intermetrics Report IR #655, Cambridge, Mass., 43p.

Hawkins, E. and Huxtable, D., "A Multi-Pass Translation Scheme for ALGOL 60," in Annual Review in Automatic Programming vol. 3, Pergamon Press, 1963, pp. 121-162.

JOVIAL (J73), MIL-STD-1589B (USAF), 06 June 1980, 168p.

Knuth, D.E., "An Empirical Study of FORTRAN Programs", Software -- Practice and Experience, 1971, pp. 105-133.

Myers, G., "The Evaluation of Expressions in a Storage-to-Storage Architecture", Computer Architecture News 6:9, June 1978.

Nebula Instruction Set Architecture MIL-STD-1862A, 1 July 1981, 169p.

Randell, B., and Russell, L., ALGOL 60 Implementation, Academic Press, New York, 1964, 418p.

Tanenbaum, A., "Implications of Structured Programming for Machine Architecture, CACM 21:3, March 1978, pp. 237-246.

Wulf, W., "Compilers and Computer Architecture", Computer 14:7, July 1981, pp.41-49.

Building Fault-Tolerant Systems with Nebula

Fred B. Schneider

September 10, 1981

References to Nebula in this report refer to the version of MIL-STD-1862A dated 1 July 1981.

Building Fault-Tolerant Systems with Nebula

Fred B. Schneider

Department of Computer Science
Cornell University
Ithaca, New York 14853

September 10, 1981

1. Introduction

A fault-tolerant system is one that can continue to execute despite malfunctions by one or more components. In this note, I comment on the suitability of using the proposed Nebula Instruction Set Architecture [1] as a processor in a fault-tolerant computing system. This note does not address software reliability issues; i.e., the ease with which correct (consistent with a formal specification) programs can be developed for a Nebula. Instead, we attempt to determine:

> Given a program that will run correctly on a fault-free Nebula, how should one construct a system using Nebula processors on which that program will run correctly despite hardware failures.

The Nebula Instruction Set Architecture does not allow for the possibility of processor failures. Although facilities are described for coping with power failures ([1] sec. 11.4) and memory system failures ([1] sec. 11.6) -- both important aspects of the support environment for a processor -- no facilities to deal with Nebula malfunctions are mentioned. This has several consequences. First, it makes it difficult, although not impossible, to construct a fault-tolerant computing

system with Nebula processors. This is detailed below. Secondly, it provides little incentive for contractors to construct Nebula processors with fault-detection capabilities. For example, if an unrecoverable hardware failure is detected, there is no well-defined action for the hardware to take. And, not all hardware failures are recoverable; some failures can be easily detected but not easily recovered from.

2. Paradigms for Building Fault-Tolerant Systems

Given almost any processor instruction set architecture, it is possible to build a fault-tolerant system by using replication and voting [2]. Suppose we desire a system that reads from an input device, writes to an output device, and can continue to function correctly despite up to k failures. Such a system can be constructed as follows. The input device is replicated 3k-fold, as is the processor and output device. Then, each processor reads a value from every input device instance as well as: the values read by every processor from each input device instance, the values every processor read from every processor as having been read from every input device instance, and so on. A processor then performs its computation based on this vector of values. Usually, the computation will be based on the median value read -- in effect, a majority consensus. Then, the results of the computation are written to each instance of the replicated output device. The action of each of the output devices is a function of all of the values written to it.

While this approach allows construction of fault tolerant systems with any specified degree of reliability, it does so at a rather substantial cost. Hardware is replicated and an extensive interconnection facility is required. Also, a good deal of actual communication is required for each computational step.

A second approach to constructing fault tolerant systems can be exploited if processors are:

(i) capable of detecting failures and

(ii) facilities exist for a processor to be stopped by program control, either locally or remotely.

Then, a system can be structured as follows. If ever a processor detects that it is not functioning properly, or that some other processor is not functioning properly, a failure interrupt is signalled on the malfunctioning processor. The effect of a failure interrupt is to invoke an interrupt handler that saves state information. Moreover, once a failure has been signalled, we assume that the processor automatically halts after some (short) length of time. This time interval should be long enough for system storage to be made consistent with the processor caches and for a reasonable amount of state information to be saved on some involatile storage medium. This ensures that a failed processor does not continue to operate in its failed mode for too long, but has an opportunity to store its state information before being halted. Whenever a processor is halted due to a failure, a reconfiguration rule is invoked to redistribute tasks over the running processors.

This approach presupposes (1) the availability of the state information necessary to resume processes that were running on a malfunctioning processor and (2) that all failures can be detected. The failure-interrupt handler provides an opportunity to store relevant state information. However, it might not always be successful, due to the nature of the hardware failure. Nevertheless, there do exist techniques for structuring programs so that state information is periodically "checkpointed" to some stable storage medium; a formal basis for this approach to program design is described in [3].

Secondly, while it is not possible to detect all failures using a finite amount of hardware, failure detection can be implemented to any desired degree by replicating hardware. For example, the Intel iAPX 432 [4] has provisions for a processor to function in "slave" mode. When operating in this mode, it monitors the input/output

bus of another processor and raises a signal level if it detects an inconsistency between its computations and the behavior of the processor it is monitoring. Raising this signal level causes a failure interrupt.

Although the Nebula specification does not presently include a failure interrupt, or a processor halt facility (remotely signalled), these features could be added with minor, if any, changes to the rest of the architecture. Then, a fault-tolerant system could be constructed, using the second approach described above with a collection of Nebula processors that can communicate by using some stable (i.e., crash resistant) storage medium. Constructing an approximation of such a stable storage system by using existing storage devices -- disks in particular -- is well within the state of the art [5] [6]. Running on top of each Nebula processor is a kernel that implements a "highly reliable" Nebula. It does this by implementing the interface as described in the present Nebula Instruction Set Architecture[1], and in conjunction with the kernels on the other processors in the system, ensuring that for suitably structured applications:

i. tasks are run (only) on correctly functioning processors, and

ii. when a task is run on a malfunctioning processor and then moved to a functioning processor, it is as if the task was never run on the malfunctioning processor.

By "suitably structured applications" we mean those that perform checkpoints of the necessary state information with sufficient frequency. (See [3].) And, while it is inappropriate to include specifications for facilities like the slave mode on the iAPX 432 in an Instruction Set Architecture, such a feature might well be provided on some Nebula implementations -- those intended for use in fault-tolerant systems.

It is useful to point out the implications of such an approach with respect to the software portability. It seems reasonable for a failure interrupt to return

[1]It is not clear whether this should include the proposed failure interrupt.

a code indicating that nature of the hardware failure detected[2]. Such a code would provide information to the fault interrupt handler about what data might be damaged and what features of the hardware should be suspect, hence not used in performing recovery. Moreover, it seems likely that these codes will be implementation dependent -- hardware error detection capabilities will vary from implementation to implementation -- and that by making use of implementation-dependent data structures, such as the exact format of a context stack, the programmer of a fault-handler will have a better opportunity to save a meaningful state. Thus, it seems reasonable to view the fault-handler as not portable; it would be dependent on the Nebula implementation and perhaps even the exact configuration of Nebulas of which it is part (to implement a reconfiguration rule). Other programs would be portable to the extent they presently are.

3. Conclusions

There is no indication that Nebula is well-suited for the construction of fault-tolerant systems. However, with only slight modification -- the introduction of a new interrupt class, called failure interrupt -- this can be changed. Lastly, the fact that many aspects of the instruction set architecture are implementation-dependent precludes construction of portable fault-tolerant software.

References

(1) Nebula Instruction Set Architecture, MIL-STD-1862A, July 1, 1981

(2) Goldberg, et al., Formal Techniques for Fault-Tolerance in Distributed Processing (DDP), Final Report Contract No. DASG60-0046 (SRI Project 7242), SRI International, May 1980.

(3) Schlichting, R.D. and Schneider, F.B. Verification of Fault-Tolerant Software, Technical Report TR 80-446, Department of Computer Science, Cornell University, Nov. 1980.

---

[2] In fact, then the failure interrupt mechanism can subsume the powerfail interrupt and memory system error traps.

(4) Intel Corp., Introduction to the iAPX 432 Architecture, Manual Order Number 171821-001, 1981.

(5) Lampson, B.W. et al, Distributed Systems - Architecture and Implementation, Lecture Notes in Computer Science Vol. 105, Spring-Verlag, 1981.

(6) Schneider, F.B. and Schlichting, R.D., Towards Fault-Tolerant Process Control Software, Proc. The Eleventh Annual International Symposium on Fault-Tolerant Computing, Portland, Maine, June 1981, pp. 48-55.

Nebula Architectural Support for Virtual Machines

Robert D. Cowles

December 4, 1981

Digicomp Research

Ithaca, NY

References to Nebula in this report refer to the version of MIL-STD-1862A as changed through 31 August 1981

Nebula Architectural Support for Virtual Machines

Robert D. Cowles

Digicomp Research Corporation

Ithaca, NY 14850

December 4, 1981

INTRODUCTION

Successful virtual machine implementations have existed almost since the introduction of the third generation computer. Early work at the IBM Cambridge Scientific Center indicated the virtual machine approach was feasible[1] even before virtual memory hardware was generally available. A subsequent implementation of a virtual machine based general purpose timesharing system[2] has proven to be quite popular and has become the major interactive operating system available across the full range of IBM 370-compatible processors.[3] Advantages of using a virtual machine environment have been known for some and include:

* measuring and testing operating system software
* running different operating supervisors or versions of the same operating supervisor at the same time
* insuring reliability and security

* making hardware changes or enhancements without requiring the recoding of existing operating systems
* running with a virtual configuration that is different from the real system (memory, different I/O devices, etc.).

In the initial efforts to select a standard military computer architecture, the selected architecture would have been required to be virtualizable if all architectures other than the IBM 370 were not excluded.[4] This paper attempts to analyze the weaknesses of the Nebula architecture with respect to its ability to support a virtual machine environment.

REQUIREMENTS FOR VIRTUALIZABILITY

For a more complete discussion of the requirements for virtualizability, the reader is referred to a well known paper by Popek and Goldberg.[5] The paper outlines several characteristics of a virtual machine environment:

1. except for timing considerations, the provided environment is essentially identical to the real machine
2. programs running in a virtual machine environment show only a minor decrease in speed
3. and the real operating supervisor has complete control of the system resources.

The major concept presented in the paper is one of sensitive instructions. An instruction is control sensitive if execution of the instruction changes the privilege mode or if it changes the memory map. An instruction is behavior sensitive if the results of its execution are dependent on the memory map or on the privilege mode of the processor.

The important result of the paper is that an architecture can be virtualized if the sensitive instructions are a subset of the privileged instructions.

PROBLEM AREAS IN NEBULA

Memory Management

The User/Supervisor map structure of Nebula presents some difficulties in simulating a Nebula machine. Certain fixed locations contain virtual addresses of procedures to be called under asynchronous conditions (timer interval expiration, I/O interruption, etc.). Since these procedures may be privileged (depending on bit 31 of the interrupt vector) and since the interrupt may not be related to the currently executing virtual machine, some intervention by the real operating supervisor is required. If the virtual machine is not allowed to enter real supervisor mode, all references to the virtual supervisor map will cause traps generating prohibitive overhead. The solution which decreases the overhead to a reasonable level is to allow the virtual machine to use real supervisor mode but make access to certain segments in the supervisor map a privileged operation. The privileged segments would contain procedures and control blocks used by the real operating supervisor to process interrupts and perform other housekeeping chores.

The use of privileged segments to contain the code and control blocks of the real operating supervisor gives rise to additional problems. The first problem is the loss of segments available to the virtual machine; if the Nebula implementation allows M segments and the real supervisor uses R segments, then only M - R segments are available to the virtual

machine. Operating supervisors that assume Q segments are available in the implementation may not be able to execute in the virtual machine provided by the real operating supervisor if Q > M - R.

Another problem with this approach arises when one attempts to choose a virtual address for the privileged segment(s). Although there is likely to be plenty of room in the supervisor virtual address space for the privileged segments, they must not overlap any segments in the supervisor map of the currently executing virtual machine. The only solution within the current Nebula architecture appears to be to relocate the privileged segments (along with any pointers to the privileged segments) to an unused portion of the virtual machine's supervisor map whenever there is a collision. The relocation can be performed fairly efficiently for code segments containing no address constants, but places severe constraints on the use of pointers between control blocks requiring that they be independent of their location in the address space.

Additionally, a virtual machine may have privileged segments which should only be accessible running in virtual privileged mode. A solution to this problem is to flip-flop the real protection status of the virtual privileged segments; i. e. make the segments privileged when the virtual machine is executing in non-privileged mode (to prevent the virtual machine from accessing them without causing a trap), and making them non-privileged when the virtual machine is executing in virtual privileged mode (to allow the virtual machine to access the segments without causing a trap).

A further problem with the User/Supervisor map structure is that the task switching instructions do not alter the cached version of the supervisor map. In the general case, to dispatch a new virtual machine (task) with the current Nebula architecture, it is necessary to perform REPENT instructions to synchronize the new supervisor map in memory with the one in cache.

Privileged Instruction Simulation

When a virtual machine attempts to execute a privileged instruction, a trap to the real supervisor should occur. If the virtual machine is executing in virtual privileged mode, the real supervisor must simulate the operation of the instruction on a real machine; otherwise the real supervisor must simulate a privileged operation trap to the virtual machine. Simulation of privileged instructions is difficult in the Nebula architecture for the following reasons:

1. Software simulation of the full set of addressing modes is quite complex and likely to entail significant overhead.
    a. The contents of registers and the values of procedure parameters are sometimes required but are not available without access to the context stack.
    b. The instruction causing the trap is possibly located in an instruction segment. The real operating supervisor may only access the operation code and operands of the instruction causing the trap by: aliasing the segment containing the instruction with a segment allowing data access; or changing the protection status of the segment using the REPENT instruction.

2. After simulation of the privileged instruction, the interrupted task must be resumed at the instruction following the one causing the interrupt. Nebula provides no convenient mechanism to resume a task other than at the point of interruption.
3. Certain instructions manipulate or create implementation dependent data and are impossible to simulate without more detailed specification than is provided by MIL-STD-1862A. In particular, the PTASK, PRAISE, and PINIT instructions manipulate the context stack pointer; and the SETSEG instruction generates implementation dependent information in the segment associators of the IOC register blocks.
4. Since all instructions have the potential of accessing a privileged segment, simulation for the complete Nebula instruction set must be handled in some fashion.

If the instruction simulation requires that a privileged instruction trap be reflected to the virtual machine, the Nebula architecture again makes this difficult. The natural operation would be to initiate a task on the virtual machine's (virtual) kernel context stack using the privileged instruction trap handler vector in the virtual machine's map. The difficulty in performing this operation centers around the inability of the TINIT or PINIT instructions to pass a parameter list. The trap procedure expects a single parameter which is a byte size reference to the opcode of the instruction causing the trap. There appears to be no implementation independent way to both create the execution context on the virtual machine's kernel context stack and pass parameters to the trap procedure.

Sensitive Instructions

Since the real operating supervisor must take quite different actions when a privilege violation is encountered depending on whether or not a virtual machine is in virtual privileged mode, the real supervisor must be able to know the virtual privilege status of virtual machine at all times. A sufficient condition for satisfying this requirement is that every instruction which can cause a change in the privilege status is itself a privileged instruction. However, in the current Nebula architecture certain instructions allow a privilege mode change without causing a privilege instruction trap. In particular, CALLU and RET (and some other RET-type instructions) can change a task between privileged and non-privileged mode. There appears to be no bypass for this problem and it is of such magnitude that if a change in the Nebula architecture is not made, execution of virtual Nebula machines on a Nebula architecture is not feasible.

Another type of sensitive instruction is one that alters location dependent control information (timer intervals, etc.). The real operating supervisor cannot allow direct access by the virtual machine to the hardware assigned physical locations, but it must be aware of any changes the virtual operating supervisor thinks it is making to those locations. (These locations are often used in vectored calls which may cause a change in privilege status, depending on the setting of bit 31 of the vector entry.) Most cases of sensitive instructions of this type can be handled by (1) making access to the first megabyte of a virtual machine's memory (I/O space) a privileged operation; and (2) intercepting all vectored calls (SVC, OPEX, etc.) to insure that the virtual machine's procedures are called in a nonprivileged mode.

The difficulty with the above technique lies in passing the proper number of parameters to the routine to be invoked within the virtual machine. It is likely to be the case that OPEX and SVC routines have the number of parameters to be passed coded at the entry address of the procedure. If the real operating supervisor gets entered for an OPEX or SVC, the virtual operating supervisor's corresponding procedure must be called (after some housekeeping functions are performed) with the original parameters. However, the original parameters may not be available unless the SVC or OPEX procedure invoked in the real operating supervisor has at least as many parameters as its counterpart procedure in the virtual operating supervisor. The real operating supervisor routines cannot really just allow for the maximum parameters since the operand pre-evaluation of arbitrary memory is likely to cause an exception. Another alternative is to change the parameter number at the entry address of the real operating supervisor's OPEX and SVC handler procedures to correspond with the parameter numbers specified in the virtual machine about to be executed -- this technique has the drawback of requiring the modification of memory locations in a procedure segment with the risk of mismatches between the instruction pipeline and memory.

Input/Output

Access to I/O Space: Access to real I/O Space cannot be allowed except possibly for devices dedicated to virtual machines (TTY's, tape drives, etc.). Accesses that a virtual machine makes to its I/O Space must be mapped into to appropriate actions to be performed in the real I/O Space.

Memory Map Restrictions: The segment restrictions affect how real memory can be allocated to each virtual machine. To satisfy the requirements for I/O segment protection, the real segments must be at least as large as the segments of the task executing in the virtual machine for which a SETSEG instruction has been issued. Of course, the real segments must have the same protection bits (instruction, data read, context, etc.) as the virtual segments. If the virtual operating supervisor runs without using the segmentation hardware, and during the initialization process immediately following an IPL or Reset, the real operating supervisor may have to provide a piece of contiguous real memory that is the same size as the memory of the virtual machine.

PROPOSED ADDITIONS TO NEBULA

Memory Management

The most painful problems in the memory management area are the problem of locating/relocating the segment containing the real operating supervisor, and the restrictions on the number of segments that a process can have.

Memory Segment for Real Operating Supervisor: This problem could be eliminated with the addition of a privileged instruction which returned information about the number of segments available on the machine, the amount of memory on the machine, and the highest unique virtual address (the implementation virtual address space). Assuming that virtual operating supervisors would issue such an instruction and/or be prepared to use no more resources (memory, segment descriptors, etc.) than was available, the real operating supervisor could reserve some of the machine

resources for itself (a few segments that only it could access for controlling resource allocation between virtual machines). The ability for an operating supervisor to issue an instruction and then configure itself based on the machine or implementation resources it is told are available greatly increases the portability of operating system code between machines.

Restrictions on the Number of Segments: Restrictions on the number of segments should be removed. Although it should be possible for the operating supervisor to find out in an implementation independent manner how many segments are implemented in the cache, the address mapping hardware should be responsible for keeping actively referenced segment table entries quickly accessible. The only penalty for executing a task with more segments than fit in the cache would be one of performance; and even that penalty would likely be removed by technology insertion if at a later date the impact appeared significant.

Synchronization of Supervisor Map: The necessity of performing multiple REPENT instructions to synchronize the cached and memory versions of the supervisor map causes unnecessary overhead. Either the task instructions should have an option which causes synchronization of the supervisor map as well as the user map; or, as recommended below, there should be new instructions for dispatching a virtual machine that include synchronization of both maps as part of their operation.

Privileged and Sensitive Instructions

Many privileged operations require little or no intervention on the part of the real operating supervisor other than simulating the instruction

by placing the appropriate results within the virtual machine's memory. Similarly, OPEX and SVC handling consists of a few housekeeping functions and then calling the appropriate procedure within the virtual machine. Given certain control blocks that the real operating supervisor maintains, it is certainly possible for much of the simulation to be performed by microcode, thereby eliminating problems with operand access and context stack manipulation which makes software simulation of these instructions impossible. Special microcode to maintain virtual privilege mode flags during the execution of CALLU and RET-type instructions would remove the problems associated with these instructions. These changes would not only greatly increase the virtualizability of the Nebula architecture, they would result in a substantial performance improvement for programs running in a virtual machine environment.

The action of the special microcode would be disabled when the machine entered real privileged state and would be enabled by when a new instruction is executed. The functions performed by the new instruction would be to perform all the functions of LTASK but in addition would set a flag indicating that a virtual machine is executing and make the supervisor map in cache consistent with memory. One possible implementation would be to use bit 2 of the PSW to indicate whether the real operating supervisor or a virtual machine is executing. If a virtual machine is executing, bit 3 could indicate its virtual privilege mode. Another approach would be to reserve an additional word in I/O space to contain a pointer to the virtual machine control block of the currently active virtual machine and use bit 31 of the quantity to indicate the virtual privilege mode. In either case, the reserved protection keys

and the reserved field in the relocation amount could also be used by microcode to substantially reduce intervention by the real operating supervisor when an instruction accesses a privileged segment.

An implemention which does not require extensive microcoding changes would be the addition of a PRIVEX vector along the lines of the OPEX vector. Any attempt to execute a privileged operation while in non-privileged mode and any other instruction which would cause a privileged mode change (like CALLU and RET-type instructions) could be intercepted and the instruction's arguments would be passed to the PRIVEX procedure (note that the number of arguments passed is defined by the instruction). Simulation of instructions accessing privileged segments would still need to be performed as described above.

Input/Output

Since the channels use virtual addresses rather than real addresses and since channel programs normally are executed from instruction segments, it appears that many problems with virtualizing the I/O have been bypassed. Privileged instructions to start and halt channel programs would be likely to greatly decrease the number of accesses a virtual machine would make to the IOC register block.

CONCLUSIONS

The Nebula architecture as it stands cannot be used to create a virtual machine environment. As an absolute minimum, microcode changes are required to the handling of the CALLU, RET-type, SVC and OPEX instructions or vectored calls. Simulation of certain instructions (e. g.

PINIT, PRAISE, PTASK) requires more specification of the context stack pointer and how it is manipulated. Additionally, microcode assist in the handling of privileged instructions and and virtual privileged segments would greatly improve the performance of a virtual machine. A special instruction (or set of instructions) to activate the virtual machine microcode assist and make both the user and supervisor maps consistent with memory when a task is dispatched would be extremely useful. An instruction which returned certain characteristics of the processor (memory size, etc.) would enhance portability of all operating supervisors, and elimination of the restriction on the maximum number of segments would enhance the portability of all programs.

REFERENCES

[1] Adair, R., Bayles, R. U., Comeau, L. W., Creasy, R. J., "A Virtual Machine System for the 360/40," IBM Cambridge Scientific Center Report No. G320-2007, 1966.

[2] Meyer, R. A., Seawrite, L. H., "A Virtual Machine Time-Sharing System," IBM Systems Journal, Vol. 9, No. 3, 1970.

[3] Cook, R., "IBM's System Software Strategy for the '80s," Computer-World, Vol. XV, No. 43, October 26, 1981.

[4] Burr, W. E., Fuller, S. H., Stone, H., Conputer Family Architecture Selection Committee - Final Report, Volume II - Selection of Candidate Architectures and Initial Screening, ECOM-4527, September, 1977.

[5] Popek, G. J., Goldberg, R. P., "Formal Requirements for Virtualizable Third Generation Architectures," Communications of the ACM, Vol. 17, No. 7, July, 1974.

Suitability of Nebula Architecture for Very High Level Languages

A. Demers

November 30, 1981

References to Nebula in this report refer to the version of MIL-STD-1862A dated 1 July 1981

Suitability of Nebula Architecture
for
Very High Level Language

A. Demers

1. Introduction

This report is an evaluation of the suitability of the Nebula Instruction Set Architecture, as described in MIL-STD-1862A [STD], for implementing very high level (VHL) languages. Such languages include the many dialects of LISP, SAIL, PLANNER, SETL, etc. The basic architectural requirements of these languages are very similar. I shall concentrate on LISP in the sequel but my comments will be sufficiently general to apply to other VHL languages as well.

In the past, artificial intelligence research has been conducted using VHL languages almost exclusively, Thus, this report can also be viewed as an evaluation of the Nebula ISA's suitability for artificial intelligence research.

I assume Nebula is a general purpose architecture. Its suitability for LISP should be compared to that of other general purpose architectures, and not machines designed specifically for LISP, such as those being marketed by LISP Machine Incorporated or Symbolics, which can be expected to outperform it in that application.

Similarly, the Standard Nebula ISA has no provision for user microcode; so its performance as a LISP machine should not be directly com-

pared to that of microprogrammable machines like the 3-Rivers PERQ or the Xerox Dorado. It seems likely that many implementations of Nebula will be microprogrammable, perhaps in conjunction with the OPEX facility, but this facility can be expected to vary greatly between implementations. For this report, I assume OPEX handlers should be written in the standard Nebula instruction set; a need for special-purpose microcoded OPEX handlers would indicate a weakness in the ISA.

This report is organized as follows. Section 2 presents some important features that characterize very high level languages and discusses the demands these features make on the underlying machine architecture. Section 3 points out some features of the Nebula ISA that make VHL language implementation more difficult and, in some cases, suggests modifications to the architecture that would improve the situation. Finally, Section 4 summarizes the report.

2. VHL Language Characteristics

Very high languages like LISP generally share two important characteristics that facilitate rapid development and modification of large systems.

2.1 The Treatment of Data

An essential feature of LISP is its "abstract" treatment of data. The fundamental data type of pure LISP, the S-expression, is quite simple: an S-expression is a list (technically, a binary tree) of values, which may themselves be numbers, symbols or (recursively) S-expressions. Production LISP systems also provide more conventional data structuring mechanisms such as arrays and records. These are incorporated into the

S-expression framework; so that, for example, a list element may be an arbitrary array or record value. Thus, an S-expression value may be arbitrarily large and, in general, no maximum size can be determined at "compile-time". Large, complex structures can be created, discarded, passed as arguments or returned as a function result, with no explicit attention to storage management. Storage allocation and automatic deallocation of unreferenced structures ("garbage collection") are built into the run-time support system. Garbage collection, in particular, takes place without program intervention.

LISP programmers argue that the convenience of not having to worry about storage management far outweighs any inconvenience that results from having to encode all data as lists. Common practice is to define a set of constructor and accessing functions that hide representation details from the programmer, thus creating a form of "abstract data type".

List-structured data and automatic storage management as described above make some fundamental demands on the host machine ISA.

(1) The architecture should support an extremely large address space. List-structured data is inefficient in its use of space - a large fraction (up to half) of the space used is consumed by pointers. There are special techniques for representing lists that alleviate this difficulty somewhat, at the expense of added execution time. Collectively these techniques have been called cdr-coding; see[Allen 78]. However, even with these techniques it is not uncommon for a large LISP program to consume several million words of storage. As program applications become more complex, storage

demands will continue to grow. For example, the current implementation of the Berkeley VAX Unix operating system, currently under consideration as a "standard" system for DARPA contractors, is considered somewhat deficient for AI research because it limits each process' virtual storage to about 6 million bytes [Joy 81].

To meet such storage requirements at the present level of technology the architecture should support virtual memory, making it possible to run programs whose storage requirements exceed the available primary memory.

(2) The architecture should support processes with large working sets. This is not immediately implied by the previous requirement, since an architecture may support a large virtual address space (i.e., its addresses may be bits long) but perform badly unless processes exhibit strong locality in their memory references (i.e., have small working sets).

Building and manipulating linked lists tend to leave list elements widely scattered in memory. Thus, LISP programs generally exhibit less locality than conventional programs, and as a result make heavier demands on a virtual memory system. Certain garbage collection algorithms involve copying and "compacting" preserved list structures to minimize working set size. An extensive bibliography on this subject appears in [Cohen 81]. Even with such techniques, however, large working set size is a serious problem with LISP programs.

(3) The architecture should support efficient garbage collection. A particularly useful approach to garbage collection involves a

concurrent process that reclaims storage continually, potentially allowing LISP to be used in real-time applications. As discussed below, the Nebula's context stack mechanism interferes with garbage collection.

2.2 Programming Environments

A second feature of LISP that has contributed to its popularity among AI researchers is the existence of usable program development environments and the ease with which these environments can be extended. LISP lends itself to the creation of such environments because, at least during initial development, every LISP program is also a LISP data object that can be examined, modified and interpreted by other programs written in LISP. A LISP programming environment can be developed and maintained entirely within LISP. A good introduction to this "subculture" is [Sandewall] 1976 .

A modern programming environment, whether for LISP, Ada or machine language, should be highly interactive and interpretive. It should provide such features as execution tracing; reverse execution; breakpoints; the ability to examine the state of a partially completed computation, change the program and/or data and continue; and many others. Typically such an environment is based on an interpreter that is prepared to handle exceptions and can examine suspended activations of itself (e.g., to produce a "stack trace") after an exception. As I discuss below, the Nebula's built-in procedure call mechanism is not suitable for this application.

3. Suitability of the Nebula ISA

3.1. Memory Management

As mentioned above, a sizeable LISP program is likely to require a large virtual address space and to have a large, scattered working set within that address space. The Nebula memory management system, based on a few variable-length segments rather than a large number of fixed-length pages, appears to be inadequate. A Standard Nebula processor may have as few as 16 segment relocation registers. Even with very large pages, a reasonable LISP program's working set is likely to contain many more than 16 pages, because list-structured data tends to be scattered rather than contiguous in memory. The obvious conclusion is that the system will "thrash", spending much of its time moving pages between primary and secondary memory rather than doing useful work.

One solution to this problem would be to run LISP only on Nebula implementations with many more than 16 segment registers. There seems to be nothing in the Standard that precludes building such machines. Unfortunately, the Nebula memory map structure is designed so that it must be searched rather than indexed into. Even if all map entries are cached, each process switch will require the working set of the new process to be faulted into the cache, involving many searches of the memory map. A sophisticated hardware implementation might do these memory map searches efficiently, using (say) interpolation search, and thereby keep the cost of process switching down to a tolerable level. However, this approach would add cost and complexity, and does not seem particularly desireable even if it could be made to work.

A somewhat more elegant approach, perhaps feasible with the current memory map design, is illustrated in Figures 1-2. Since the Nebula's memory management scheme is discussed in some detail elsewhere, the approach is simply sketched here.

Operating system software maintains a "conventional" page table, which is accessed by indexing into it with some number of leading bits of a virtual address. The hardware memory map is used as a cache for a few of the most recently used page-table entries. Pages are classified as:

(1) resident in memory and in hardware memory map;

(2) resident in memory, not in hardware memory map; and

(3) nonresident.

An attempt to reference a page in category (2) causes an **Invalid.Segment** trap (or an **Invalid.Access** trap - see below). The operating system responds to such a trap by replacing the same hardware map entry by the entry for the requested page. This scheme is similar to the one used by VMS and Berkeley VAX Unix to make up for the absence of "referenced" bits in the VAX address translation hardware [Babaoglu 80].

The Nebula memory map design assigns contiguous virtual addresses to segments. The only apparent way to map noncontiguous segments is to separate them by "dummy" segments with access control bits set to 000 (no access). Figure 2 illustrates this. Note that about half the memory map registers are wasted. I expect that 8 map registers (all that would be available on a minimal Standard Nebula using this scheme)

are not enough to give acceptable performance.

Possibly the designers of the Nebula memory management system had some other more clever scheme in mind, or perhaps they do not believe in paging. In any event, it would be comforting to know, through analysis or simulation, that the Nebula can be made to work when the virtual address space is much larger than physical memory.

3.2. Exceptions and Interactive Debugging

The interpreter in an interactive programming environment must be prepared to handle certain exceptional conditions itself. For example, an **Illegal.Divisor** trap by default should put the system into interactive debugging mode, in which the programmer can examine the state of suspended procedures, modify procedures and data, and perhaps attempt to resume execution.

With this application in mind, I am puzzled by the Nebula's exception handling architecture. When an exception occurs and the current procedure's exception handler state is "defined", the exception handler is entered by a direct GOTO, without saving the program counter [STD.p33]. This makes it difficult to resume execution from the point of the exception.

With cooperation from the supervisor exception handler, a "resume program counter" can be made available to the current procedure exception handler as follows. The UDLE bit of the PSW should be set, so that the exception is passed to the supervisor exception handler. The supervisor handler should save its program counter arguments at an agreed-upon memory location and pass control to the procedure exception handler

(using ERP). While this solution seems to work, such a "ritual dance" should not be necessary.

Interactive debugging presents another interesting problem. It is important for the debugger to be able to examine, print and modify the suspended state of the computation. For example, a ubiquitous feature of interactive debuggers is the ability to print a "call trace" - a list of procedure calls (with their arguments) leading to the current state. This will require examining and changing previous contexts on the context stack. However, as I discuss in the next section, the Standard effectively prohibits this.

3.3. The Context Stack

3.3.1. A Problem

The Nebula ISA includes a "structured" procedure call mechanism with hardware-maintained procedure context stacks. Parameters, return addresses, saved registers, etc., appear in the context stack in an undefined format. Parameters are accessible only by means of special addressing modes, and only the parameters of the currently active procedure are accessible at all [STD.8.1.3].

It has been remarked elsewhere that this design prevents nonlocal references to parameters on the context stack, even though such references are legal in Ada (or any other language with internal procedure declarations).

A partial solution to this problem is to analyze the source program to determine which parameters are non-locally referenced, and to pass such parameters on the SP stack rather than on the context stack. Of

course, such parameters cannot be referenced using the built-in parameter addressing modes; apparently the Nebula's architects assume that most parameters are referenced only locally.

LISP uses dynamic rather than static scope; that is, a non-local reference to 'x' is bound to the most recent instantiation of anything named 'x', rather than to the most recent instantiation of the lexically enclosing 'x'. Because of this scope strategy, almost every LISP parameter is at least potentially the target of a nonlocal reference. To see this, consider any function f that contains a call to an externally defined function g:

$$f = \lambda x. \quad \ldots(g\ 17)\ \ldots$$

By examining only f, it is not possible to determine whether its parameter x is non-locally referenced. If g is defined by

$$g = \lambda y.\ x$$

then the call of g from within f results in a non-local reference to the parameter x of f, <u>independent</u> of where the declaration of g occurs in the program. The existence of a nonlocal reference depends on whether g (or any function actually called by g) contains a non-local reference to a variable named 'x'.

List allows individual functions to be compiled, and allows compiled and interpreted functions to call one another freely. Thus, when compiling a LISP function, it is not possible to identify and examine all the other functions it might call, to decide whether a parameter can safely be placed on the context stack. Consequently, all LISP parameters (or at least copies of them) must be kept in user memory rather

than on the context stack.

A related reason not to use the context stack concerns garbage collection. A typical garbage collector operates by "marking" all accessible objects, and then reclaiming the space occupied by inaccessible (i.e., unmarked) objects. It does this by first marking all objects referenced directly by any of the variables in some "known" set; then marking all objects referenced by any of the newly-marked objects; and repeating this process until no more objects can be marked. Garbage collection may be initiated by actually running out of storage, or (in real-time systems) may be performed by a concurrent process.

The set of "known" variables referred to above must include the parameters of suspended function calls - it is easy to construct examples in which an object cannot be discarded, but the only reference to it is from a parameter of a suspended procedure. For example, consider the following LISP-like program:

$f = \lambda x$. **cond** [x=0- 0 ; **t- cons**[f[x-1];f[x-1]

$g = \lambda y$. **cons** [f[11]; y]

g [f [17]]

The call f[17] builds a (large) binary tree that is passed as an argument to g, thus bound to the parameter 'y' on the stack. Suppose garbage collection takes place during evaluation of f[11] (called from g). Clearly the object bound to y cannot yet be reclaimed, since it is part of the result of the call to g; but the only reference to it is the suspended parameter 'y' itself. Thus,

(1) the marking phase of the garbage collector must examine parameters on the stack; and

(2) the Standard explicitly prohibits examining parameters in the context stack;

and again we conclude that the context stack parameter mechanism is not suitable for LISP implementation.

3.3.2. A Proposed Change

The Nebula architecture needs to be extended so a process can examine and modify suspended procedure contexts. This should be done without violating protection constraints and, if possible, without doing great violence to the architecture.

A simple change would make the following possible in privileged mode:

(1) Flush the context stack cache. This is currently possible in a backhanded way using LTASK and STASK.

(2) Given a pointer to a context, produce a pointer to the previous context.

(3) Given a pointer to a context, decode, read and/or write its i^{th} parameter specifier.

(4) Given a pointer to a context, decode, read and/or write its i^{th} saved register.

This could be accommplished in two ways. One way would be to modify STD,8.1 to specify parameter and register representations in the context stack. The access control mechanism of the memory mapping system

would be used to restrict access to the context stack, as is done now. The other way would be to introduce new (privileged) instructions. This approach has the advantage that the representation of contexts in memory could remain implementation dependent.

These changes have the advantage of giving an upward-compatible extension of the current architecture. Their principal disadvantage is that references to previous contexts require assistance from the Supervisor and thus are relatively inefficient.

For reasons of efficiency, it would probably be unacceptable to use this mechanism as the standard one for non-local parameter references in an Ada implementation. However, for less demanding applications like an interactive debugging package, the mechanism would be quite adequate. In combination with, say, a shallow binding LISP parameter passing implementation, which would immediately copy every parameter from the context into an optimized data structure in user memory [Baker 78] it could yield an acceptable LISP implementation.

3.4. Preserving Register Values

A common technique for gaining efficiency in a tightly-coded program is the use of registers to hold global quantities - for example, the instruction and argument pointers in a LISP interpreter. Similarly, the run-time system in an Algol-like language implementation may dedicate a register for use as a display pointer, etc. This use of "register global variables" is not supported by Nebula, since the only register whose value is inherited by a called procedure is the SP itself. The Nebula's built-in procedure calling mechanism does not require

inherited register values; but for applications that must depart from the "standard" procedure calling conventions, inherited register values may be quite important.

The motivation for not inheriting register values seems fairly clear. The semantics of registers in Nebula corresponds to their easiest possible implementation: they are simply locations in memory near the top of the context stack. Easy implementation does not in itself justify a feature, however. Allowing register values to be inherited would add flexibility to the architecture, and should be considered.

3.5. Instruction Set

On the whole, the Nebula instruction set seems adequate for LISP. Several points deserve specific mention:

1. Auto-increment and auto-decrement addressing modes are not included. However, these addressing modes are not particularly useful in processing LISP's linked-list data structures. Their principal use is in dealing with contiguously allocated structures (strings, arrays, records, etc.), and in most cases the Nebula's string operations should work better. An elaborate LISP implementation might be able to use auto-increment/auto-decrement addressing to process a cdr-coded list, but the potential benefits do not seem great.

2. Surprisingly, the Nebula addressing modes do not allow indirection through a memory location. For LISP this is not a great loss. The facility might be useful for taking car of a list to which the only reference was in memory - cdr would of course require indexing. However, addresses of list structures currently being manipulated tend to

be kept in registers rather than in meory; and both register indirect and register indexed addressing modes are provided.

3. The principal deficiency of the instruction set seems to be the inability to stack/unstack a set of registers in a single instruction. As discussed in point (2), registers are important in the Nebula, since the addressing modes treat them quite differently from arbitrary memory cells. Saving and restoring of multiple registers on the context stack is accomplished in a single machine instruction by the built-in procedure call mechanism. However, for the reasons discussed above, the context stack is inappropriate for use by a LISP interpreter. Thus, registers must be saved/restored individually, a time-consuming process.

4. Conclusions

Obviously LISP and other VHL languages can be implemented on the Nebula. However, the salient "structured" features of the architecture -most notably its context stack procedure call mechanism - are inappropriate for such languages. The problems with non-local parameter references and garbage collection are less serious for Ada than for LISP, but still exist. I believe the architecture could be considerably improved by some simple additions and modifications. Specifically,

1. While Nebula has adequate address space, its memory mapping scheme appears inadequate for programs whose virtual memory requirements exceed the available physical memory. It is important to know whether some "clever" paging scheme like the one described above would give acceptable performance with only the 16 memory map registers required by the Standard. Simulation results would certainly suffice, and may already

exist.

2. The most important deficiency of the Nebula ISA is the inability to examine suspended procedure contexts. This problem could be lessened by defining the representation of parameters and registers in the context stack and providing a way to flush context stack cache, as discussed above.

3. Many applications could benefit from a more flexible treatment of the registers. In particular, register values should be inherited by called procedures, and instructions to save/restore sets of registers on the SP stack should be provided.

5. References

[Allen 78]
Allen, John. Anatomy of LISP. McGraw-Hill, 1978.

[Babaoglu 80]
Babaoglu, O., William Joy and Juan Porcar. Design and Implementation of the Berkeley Virtual Memory Extensions to the UNIX Operating System. Computer Science Department, U.C. Berkeley.

[Baker 78]
Baker, H.G. Jr. "Shallow binding in LISP 1.5." ACM Communications 21:7, July 1978.

[Cohen 81]
Cohen, Jacques. "Garbage collection of linked data structures." ACM Computing Surveys 13:3, September 1981.

[Joy 81]
Joy, William and R. Fabry. Proposals for enhancement of UNIX on the VAX. U.C. Berkeley, July 1981.

[Sandewall 76]
Sandewall, E. "Programming in an interactive environment: the LISP experience." Linkoping University, Sweden, 1976.

[STD]
MIL-STD-1862A. Nebula Instruction Set Architecture. May 1980.

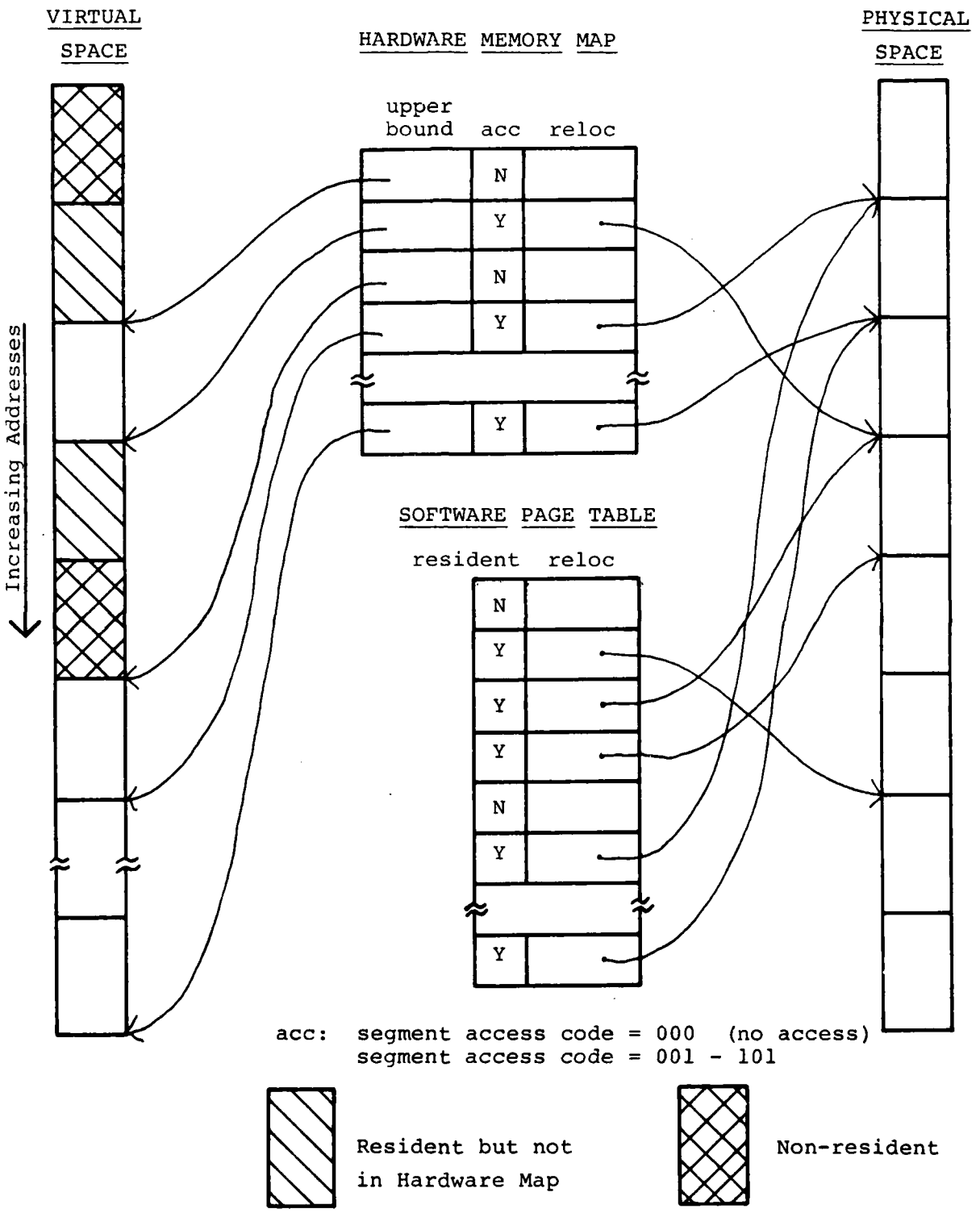
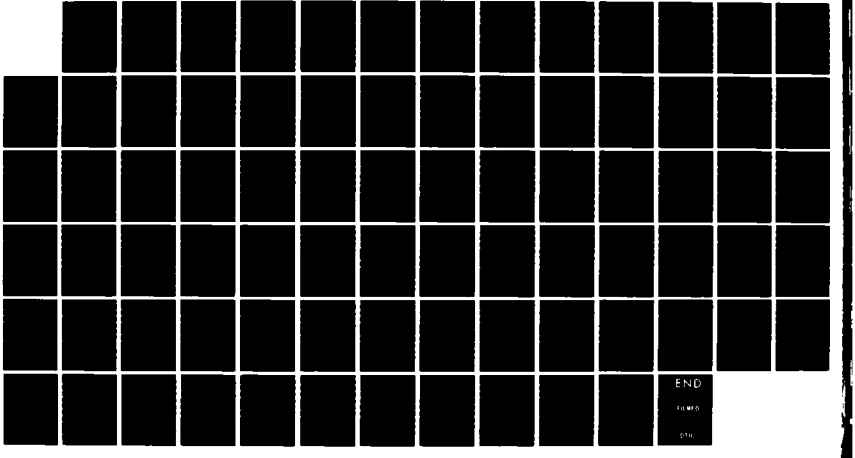
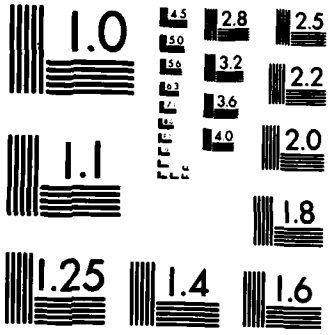

VIRTUAL
SPACE
HARDWARE MEMORY MAP
PHYSICAL
SPACE
upper
bound
acc
reloc
N
Y
N
Y
Y
Increasing Addresses
SOFTWARE PAGE TABLE
resident
reloc
N
Y
Y
Y
N
Y
Y
acc: segment access code = 000 (no access)
segment access code = 001 - 101
Resident but not
in Hardware Map
Non-resident

Figure 1

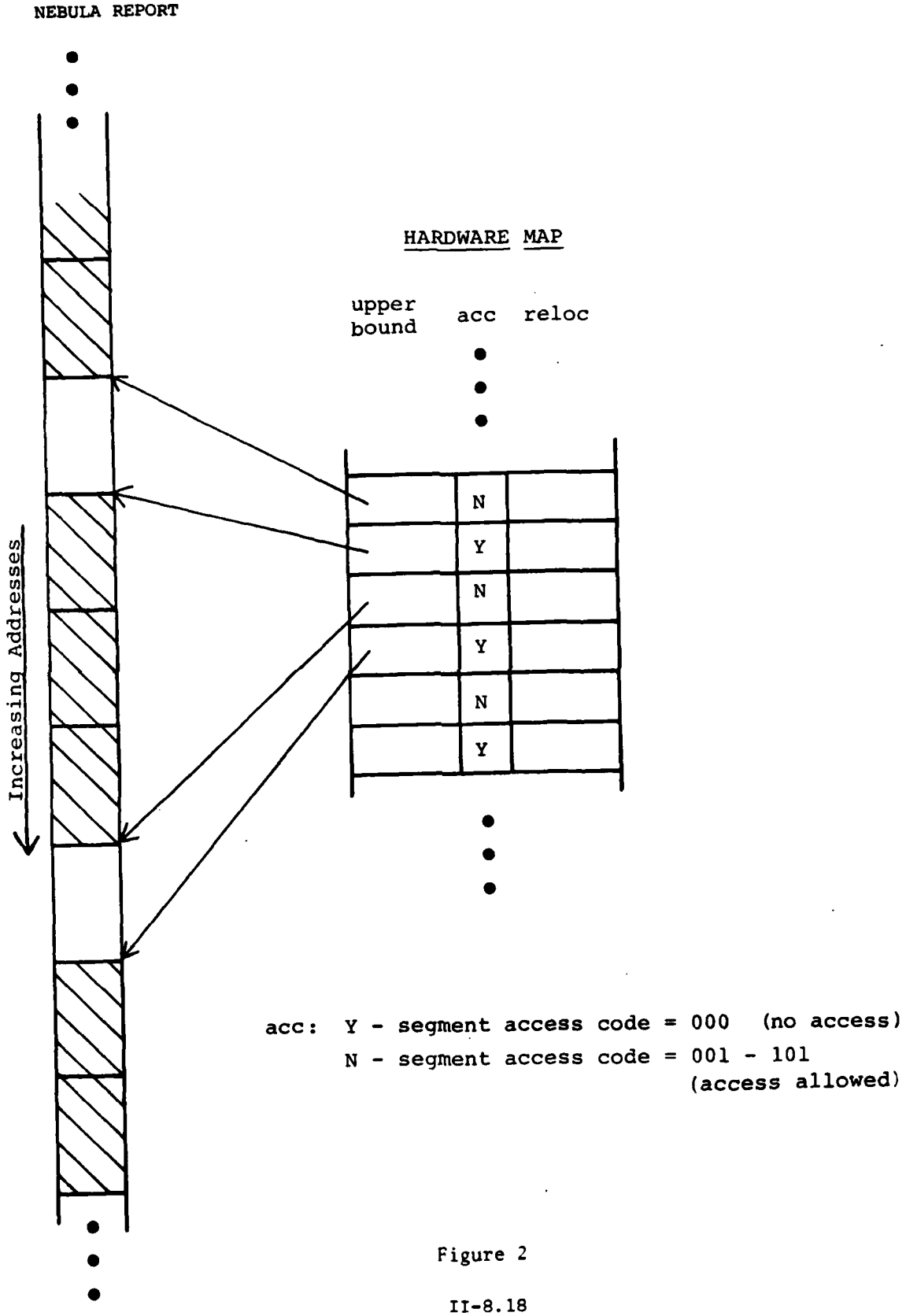
AD-A151 041 NEBULA INSTRUCTION SET ARCHITECTURE (ISA) EVALUATION 4/4
(U) DIGICOMP RESEARCH CORP ITHACA NY R D ARNOLD ET AL.
SEP 84 RADC-TR-84-190 F30602-80-C-0279

UNCLASSIFIED F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A



acc: Y - segment access code = 000 (no access)
N - segment access code = 001 - 101 (access allowed)

Figure 2

Evaluation of the NEBULA Processor

for the

Implementation of Database Management Systems

Haran Boral
David J. DeWitt

References to Nebula in this report refer to the version of MIL-STD-1862A dated 1 July 1981.

1. Introduction

A database is a collection of time varying information that models some segment of the real world. A database management system (DBMS) is that collection of programs that provides access to a database in a manner that guarantees the consistency of the stored data. An application program is a user written query or update program that interacts with the database through the DBMS. The DBMS is viewed by an application program as a virtual machine.

Implementation of DBMSs has been an active subject of research. Although many database management systems, both experimental and otherwise, have been implemented, little is known either about the implementation experiences or the operating system and hardware support required by a DBMS. Most of the known information is speculative and/or based on sporadic experiences.

In this report we discuss the suitability of the proposed military standard Nebula computer for the implementation of DBMSs. Our evaluation of the architecture is based on the document MIL-STD-1862A titled "Military Standard NEBULA Instruction Set Architecture". In Section 2 of this report we characterize databases and DBMSs in some detail. Section 3 contains a discussion of the implications of the proposed instruction set on the implementation of DBMSs. In Section 4 we discuss the implication of the machine organization, particularly the virtual memory and the I/O sub-system. In Section 5 we describe the features that an operating system must provide in order to permit the successful implementation of a DBMS on the Nebula computer. Section 6

provides a summary of this report.

2. Background Information

2.1. Characterization of Databases

There are several criteria that can be used to characterize databases and the programs that manage them. Our purpose here is to indicate to the Nebula machine designers the type of the data to be stored and the type of actions to be executed on the data. We therefore will consider the following points in characterizing databases: database type, database size, and access patterns to the database.

By database type we mean the type of the data items that are accessed and manipulated by the DBMS software and the application program. An example is numeric data. By access patterns to the database we mean the frequency of each type of access, e.g. numerous short updates and few long retrievals.

Most existing databases are formatted. That is, they consist of many records each of which is logically viewed as a fixed length string. Examples of formatted databases are business databases such as those used in the banking and airline reservation systems; statistical databases such as the census database; picture databases such as those used to store satellite photographs; and databases used to store graphical information in design automation systems. Unformatted databases typically contain documents, such as law briefs.

Some databases contain only (or almost only) data of one type. For example, a VLSI design database will contain only

geometric objects, a statistical databases will contain numeric data almost exclusively, and a business database will contain both numeric and textual data.

Database sizes vary considerably from one database to another. For example, to store the results of a small statistical survey only a few thousand bytes will be required. On the other hand, a census database will require hundreds of reels of tapes for storage.

Finally, regardless of the data model employed by the DBMS there are a number of operations, both update and retrieval, that require multiple scans of data segments and other operations that require only a single scan. Furthermore, under certain conditions it is known ahead of time what the reference string (or part of it) is. Such knowledge can be used to intelligently fetch data from mass storage and cache it.

In summary, databases of many different sizes and data types exist. Except for some database types, most databases will have both numeric and non-numeric data. Most DBMSs incorporate knowledge about the behavior of the programs to intelligently access data on mass storage. Other than operations on specific data types (such as add two integers or compare two strings), which the architecture might support directly, the differences between the database types are apparent only at higher levels, e.g. the operating system or the database management system itself.

2.2. Characterization of DBMSs

This discussion is based on our experiences with the implementation of a CODASYL DBMS here at the University of Wisconsin, discussions with members of the groups that designed, implemented, and evaluated the performance of the relational DBMS INGRES at Berkeley and the relational DBMS System R at the IBM San Jose research facility.

All three systems are fairly large, requiring between 100 Kbytes and 1 Mbyte of main memory to run. However, of this code only a small amount is heavily used. Measurements of one version of INGRES have indicated that most of the system execution time is spent in a very small portion of the code - about 10 Kbytes. This fact served as one of the design bases for the Britton Lee database machine. Our experience with the QQQ DBMS here at Wisconsin is similar.

Both INGRES and QQQ are not full fledged DBMSs and serve mainly university user communities. System R, on the other hand, implements all the data management functions required of a commercial DBMS. It's busiest component is the Research Storage System (RSS) which contains the access methods of System R. The present implementation of the RSS uses approximately 250 Kbytes.

Since most of the code in a DBMS is used infrequently, DBMSs are structured as a collection of segments, some of which are resident in main memory for long periods of time, while others are brought in on demand and rarely.

3. Implications of the Proposed Instruction Set

In the previous section we have shown that databases can contain data of several data types, in particular numeric and textual. The Nebula instruction set seems to be well suited for processing data of both types. There are, however, a number of comments that we wish to make about the string processing instructions (MOVBK and CMPBK).

In order for database systems to efficiently support access and manipulation of formatted and unformatted textual data, an efficient implementation of both these instructions is very important. Their implementation should permit the comparison of two records, or two fields within the same or different records, in the minimum amount of time. The Nebula manual specifies that two strings are to be compared item by item. While this is the correct logical view of comparing two strings, the physical implementation need not adhere to it.

There are several improvements that could be made. For example, the microcode could fetch data from the main memory in blocks, where a block is defined to be the maximum number of bytes that can be transferred from the main memory to the processor in one operation. The microcode would still compare the two strings item by item, but if a "wide line" existed from main memory to the processor, multiple items could be compared for each memory access performed.

Currently, there are three string instructions: CMPBK, MOVBK, and MOVM. The COMBK instrcution essentially will check for the equality of two equal length strings. A side effect of

the instruction is a check for inequality. In order to facilitate the efficient implementation of string operations the COMBK instruction should:

(1) Allow for the application of each of the six relational operators (equal, not equal, less than, less than or equal, greater than, and greater than or equal).

(2) Allow any of these operations to be applied to strings of varying lengths. This is particularly important for unformatted databases. We believe that one implication of text and word processing by computers is that more support will be required of computers for string operations, particularly on varying length strings.

(3) If the string instructions are to remain unchanged, one simple, and helpful, modification would be to have the COMBK instruction return the position of the first nonequal bytes in the two strings compared.

The MOVBK instruction should also be implemented in such a way as to process as many bytes as possible in each loop execution of the microcode.

Finally, although many alternative concurrency control strategies have been proposed for database systems, each mechanism requires the ability to control access to some code in a critical section. The SETBIT instruction seems to be an adequate low-level primitive for this purpose. While it might be convenient to have a higher level concurrency control primitive in the

instruction set, the "optimal" concurrency control mechanism seems to vary with the type and size of the database being accessed. We do not see a need for a large number of indivisibly executed instructions.

4. Implications of the System Architecture

The following discussion is predicated on two assumptions: there will be an ample amount of main memory in any Nebula configuration; and, segments are defined dynamically by the operating system, perhaps based on previous usage of the code.

4.1. Memory Management

In Section 2.2 we pointed out that a DBMS (regardless of the data model used) can be implemented as a collection of segments, one of which consists of a small part of the total amount of the code and is the most heavily used. The organization of the virtual memory system in the Nebula computer lends itself very well to this type of an implementation. One segment can be defined to be that procedure which is the most heavily used. The operating system can guarantee that this segment remains resident in memory at all times. The more exotic functions provided by the DBMS can be stored on mass storage. When a call to one of these routines is made, the DBMS can inform the operating system exactly which routines should be brought in (it may be the case that a call to one of these routines will involve internal calls to a number of others). The operating system can then, dynamically, form a segment, fetch the routines from mass storage, and bring the segment into main memory.

The nicest feature of the memory management system is that once a segment has been brought into main memory, accessing it is very cheap because of the simple mapping scheme from a virtual address to the corresponding physical address. Having a large

amount of main memory would enable the operating system to keep the heavily used portions of code in memory for long periods of time taking advantage of this feature.

4.2. Limitations of the Memory Management System

A user interacting with a DBMS will, during the lifetime of a single transaction (application program), be accessing data in several files. Ideally, in a computer that uses segmentation, one should assign each data file to a segment. Clearly, this is not a possibility for the proposed Nebula architecture because of 1) the need to bring a segment into memory in its entirety, and, 2) the small number of segments that a user task can have associated with it. The consequence of this design decision is that the DBMS is forced to perform several functions that would otherwise be performed by the operating system (e.g., buffere management). Nowadays, all DBMSs duplicate various operating system functions. Researchers have recently recongnized this as a problem and have only begun to address some of the issues (see Section 5.2).

If each user task could have associated with it a large number of segments then it will be possible to implement a database management system that uses one segment for each file or relation. In principle the number of segments should be unbounded, although since at any given point in time only a few segments will be used, the number of registers that hold segment addresses need not be larger than say 100. Such an approach can lead to a very clean database system implementation but will be

impossible in the currently proposed Nebula architecture since segments are not paged.

4.3. Data and Code Sharing

An important service needed by a DBMS is the ability to share both code and data. In general, this is a capability provided by the operating system and we shall discuss it in more detail in Section 5. The memory management hardware facilitates sharing of segments through the capability of mapping two segments to the same physical address.

4.4. I/O Subsystem

The I/O subsystem of the Nebula architecture is very well designed and should not present any problems with regard to database system implementation. In order for the Nebula to interface with other computers in a distributed database environment or with a database machine must have a communications subsystem. The sophisticated channel capability of the Nebula processor makes it possible to place much of the communications software in the channel itself. This should significantly reduce the burden placed by communications protocols on the host.

5. Impact of the Nebula Operating System

While control of an operating system for the Nebula processor is quite obviously beyond the control of the machine architects, many of our earlier comments on the design of the instruction set and systems architecture with regard to the implementation of a DBMS would loose a great deal of significance if the

operating system implemented on the Nebula processor did not provide certain features. In this section we describe the operating system support that we feel is crucial for the successful implementation of a DBMS. The implementors of an operating system for the Nebula processor are strongly encouraged to read [Stone81].

5.1. Efficient Inter Process Communication (IPC) Facility

A basic mechanism that all IPC facilities provide is a means by which a process can receive information (e.g. queries or results) from other processes on the same or different processors. Generally, this facility provides the process with the ability to declare a publically addressable name corresponding to a logical communications channel that it listens to for requests. While the UNIX pipe facility provides a similar facility, pipes only provide a mechanism by which "related" processes (i.e. father-son or son-son) can communicate, not unrelated processes.

Efficient IPC communication facilities are important to DBMS implementation for a variety of reasons. Their most important role has to do with the implementation of the database system itself. In Figure 1 the structure of INGRES, a relational DBMS, is shown.

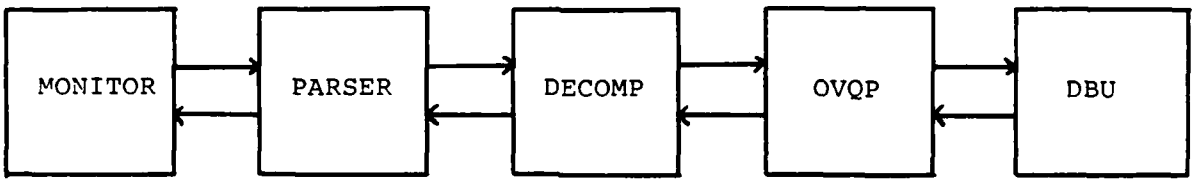


Figure 1
Process Structure of PDP-11 INGRES

Because of the limited address space provided to a process by the PDP 11 architecture, it was necessary to divide the INGRES software into five communicating processes. As shown in Figure 2 INGRES on a VAX 11/780 runs as two processes.

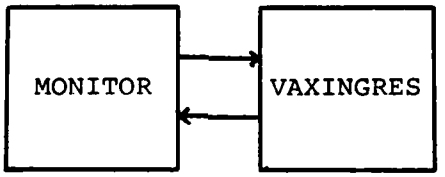


Figure 2
Process Structure of VAX 11/780 INGRES

The MONITOR process acts as a terminal monitor to provide a high-level user interface. The second process, labeled VAXINGRES, is responsible for query parsing and execution. Note that while

all five INGRES processes could have been compressed into a single process on the VAX 11/780 it was left as two processes so that the MONITOR process could be replaced with a user program (or perhaps an alternative user interface) as shown in Figure 3.

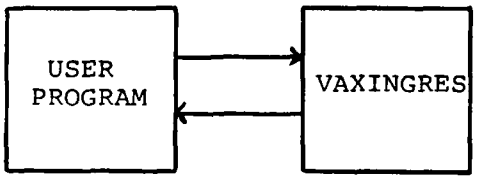


Figure 3

Another application of IPC facilities in database system implementation is shown in Figure 4. This figure depicts the organization of QQQ. Each user program that is accessing the database corresponds to one user process. When a user initiates a database call it is sent to the QQQ process for execution. By centralizing all database access in one process that is accessed through an IPC mechanism we were able to significantly simplifying buffer management and concurrency control in the database system.

Finally, there are two other important applications of IPC mechanisms in database systems: database machines and distributed database systems. In both these cases it is crucial that processes on different machines be capable of communicating each

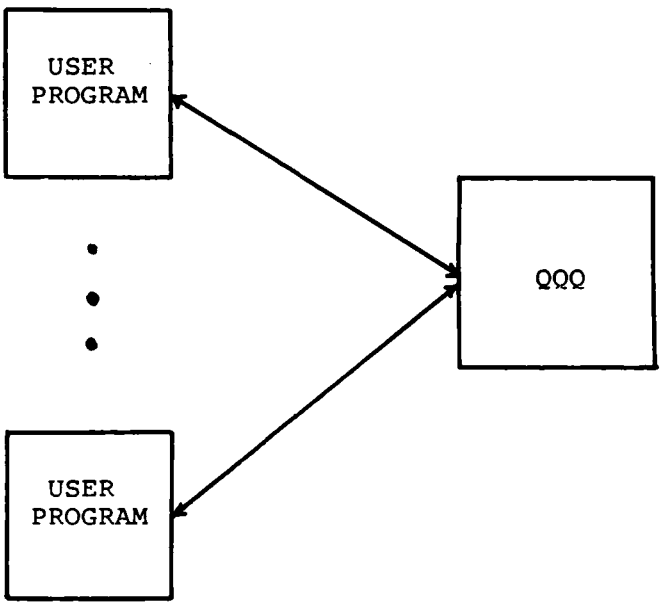


Figure 4
Process Structure of QQQ

other in a straightforward manner. Each Nebula processor that is a node in a distributed database system must be able to communicate with other sites in order, for example, to execute a query that references data that is distributed at multiple sites. If a Nebula processor is used as a front-end host to a database machine, then in order for the database machine to return results to the user that submitted the query, the process corresponding to the user must have a "name" to which that the database machine can send results.

5.2. Duplication of Effort

One problem that frequently arises in attempting to implement a database system is that the operating system does not use the appropriate strategy in providing a service to the database system. The consequence of this is that the database system designer is forced to provide the appropriate function in user space. A typical example is that of buffer management for data pages. Most operating systems manage Input/Output page buffers (data brought in from a disk) with a Least Recently Used (LRU) strategy. While the LRU strategy may be an excellent strategy for many applications it is a very poor strategy for the access patterns displayed by some database operations. Ideally, the operating system should accept "hints" from the database system on how to manage the buffer pool. The DBMS has some knowledge about how the blocks in a file will be referenced based on the user query and the access method (e.g. secondary indices) being used to execute the application program.

5.3. File System Organization

The operating system component that is perhaps most crucial to the implementation of a successful database management system is the file system. There are two popular approaches to file system design. The first is the "UNIX-style" approach in which a file is viewed simply as a character array of dynamically varying size. The second approach, of which the IBM VSAM organization is an example, is for the file system to understand the concept of a record and provide a structured file with multilevel directories,

hashing, and secondary indices to the database system. As we will discuss below, a database system needs this second type of file system. Furthermore, implementing this type of file system on top of a character array object is generally not very efficient.

Two problems arise when a database management system attempts to provide a structured file organization on top of a character array object. The first has to do with physical contiguity of blocks. Since character array objects are normally expanded a block at a time, the result is that the blocks comprising a file are physically scattered across the entire disk volume. Hence logically adjacent blocks in the file are almost never physically close (ie. on the same track or same cylinder). Since database systems frequently perform sequential scans of an entire file, the cost of such a scan (due to increased head movement) is significantly higher than if the blocks were stored physically adjacent to one another.

A second problem that arises when a database management system attempts to provide a structured file organization on top of a character array object is that multiple levels of tree structures are introduced. For example in UNIX, the blocks that comprise a file are kept track of in a tree (of indirect blocks) which are themselves pointed to by a block (the i-node). A database system that constructs a multilevel directory structure (ie. an ISAM file) on top of this organization adds yet another level of tree structure. Clearly one tree structure that directly maps a structured record-oriented onto a mass storage will be significantly more efficient.

Finally, the file system should support a varying block size for data transfers. This would permit database systems which frequently deal with large volumes of data to transfer a track at a time while other application programs could transfer data in smaller units.

5.4. Data Sharing

At a given point in time there can be several users interacting with a database through the DBMS. It is useful in many instances to permit the user processes to share data with each other as well as with the DBMS. As pointed out in Section 4 the memory management system can support this data sharing. However, this capability must also be supported by the operating system through the implementation of semaphores or similar measures that guarantee data integrity. Such a facility is lacking in a number of operating systems, for example UNIX.

6. Summary

In this discussion we have presented our views on the suitability of the Nebula architecture for implementing a database management system. In general, the instruction set and systems architecture should not present any severe obstacles to such an implementation. In terms of the instruction set, the most crucial thing that we uncovered was the fact that the CPMBK instruction did not permit for arbitrary comparisons of two character strings. The main limitation of the system architecture of the Nebula processor appears to be the fact that since segments are not paged the maximum size of a segment is limited to the

physical memory size of the processor. This prevents one from designing a database management system in which each relation, for example, corresponds to a segment. While this is not necessarily critical it does limit the options of the database systems architect. We have also provided a broad overview of the types of services required from the operating system.

At about 1986, the time the Nebula is expected to make its entry to the market, we believe that many of the data management activities handled by general-purpose computers today will be off-loaded to database machines. However, some data management functions will be implemented on general-purpose computers. This is particularly true for small and medium size databases. Our evaluation does not include a discussion of the inter-processor communication facilities in the Nebula for various reasons. However, this will be an important feature of machines at that time.

7. REFERENCES

[Stone81] Stonebraker, M. "Operating System Support for Database Managment", ACM CACM, Vol. 24, No. 7, July 1981.

The Nebula Architecture and Multiple-Processor Systems

Dr. Marvin H. Solomon

November 5, 1981

References to Nebula in this report refer to the version of MIL-STD-1862A dated 1 July 1981.

THE NEBULA ARCHITECTURE AND MULTIPLE-PROCESSOR SYSTEMS

by Dr. Marvin H. Solomon
Computer Sciences Department
University of Wisconsin
1210 W. Dayton St.
Madison, WI 53706

1. INTRODUCTION

This report contains a study, comissioned by Digicomp Corporation, of the Nebula architecture [1] as regards its suitability for multiple-processor configurations. Throughout this document, "Nebula" refers to the instruction-set architecture described in MIL-STD-1862A, 1 July 1981. The opinions expressed herein are solely those of the author and do not necessarily represent the University of Wisconsin or Digicomp Corporation.

Multiple-processor systems are often classified as "tightly-coupled" and "loosely-coupled". However, these terms are used in very different ways by different authors. Therefore, I will use the following classification instead:

A computer network is a collection of largely autonomous processing systems connected by a communications network. Nodes may be very different hardware and software systems, usually designed independently, with communications hardware and software added "on top". They may be geographically distributed over distances ranging from a few meters (local-area networks) to many thousands of kilometers. Communications bandwidth usually ranges from 10K to 10M bits/sec for local area networks and from 300 to 56K bits/sec for geographically distibuted networks. Limitations of bandwidth and delay effectively limit the frequency of in-

teractions between process on different nodes to the range from one to 10^{-6} per second (that is, from once every few seconds to once every few days). I shall not have much to say about computer networks in this study.

A multicomputer also consists of independent computer modules connected by a communications medium. It may be homogeneous (all nodes the same) or heterogeneous in hardware, software, or both. Nodes must be physically close (same room, same rack, or same board). However, it differs from a local area network in two important ways: First, the communications hardware and protocols must be designed to keep both communications overhead and delay within the bounds necessary to allow interactions at the rate of 10^3 to 10^6 per second. Second, it is designed as a unified system to provide the same sort of facilities as are provided by a single processor, but with higher throughput, shorter delay, or greater reliability.

A multiprocessor consists of several processors sharing a common memory. Here, very close interaction is possible (upwards of 10^6 per second). Multiprocessors may be MIMD (multiple instruction stream/multiple data stream--processors operate asynchronously), SIMD (single instruction stream/multiple data stream--processors operate in "lock step" on different parts of memory) or pipelined (which might be called MISD). I shall concentrate on MIMD multiprocessors.

Of course, the distinctions are not sharp and many hybrids are possible. For example the EDEN project at the University of Washington is investigating ways in which personal computers in a

local area network can be used, when otherwise idle, as part of a multicomputer. Cm* at Carnegie-Mellon University [2,3,4,5] is a multicomputer, but a memory mapping function in the interconnection network allows the memories of the component modules to be addressed as if it were a shared common memory.

In this report, I will confine myself to considerations affecting the suitability of the Nebula architecture for multiprocessors and multicomputers, although many of the same considerations that concern multicomputers also concern networks.

2. MULTIPROCESSORS

In a multiprocessor, multiple processing elements access a common random-access memory. The memory may be multi-ported or may be accessed by a shared bus. The processing elements may be "bare" procesors or they may each have a local memory cache and/or local memory visible at the architcture level. Clearly, the aspect of the Nebula architecture most directly related to multiprocessor configurations is the memory mapping scheme.

Memory mapping presents to the processor a "flat" virtual address space of 2^{32} bytes. This space is divided into two halves by the high bit of the address, and each half is further subdivided into several "segments". Each address up to some maximum in each half lies in exactly one segment; that is, there are no "gaps" between segments. Each segment is independently protected and mapped to a block of physical addresses. The number of segments in each half is implementation-dependent, but

the standard requires at least 16 segments in each half. Segments may be divided on any 8-byte boundary (in both virtual and physical space), although the standard allows an implementation to coarsen the grain up to 256-byte boundaries. Mapping for either half may be turned off, so that virtual addresses (with the high bit stripped) are treated directly as physical addresses. The map for each half is descibed by a table. The physical address of this table is stored in a fixed physical location (FF824 for the high (supervisor) half and FF828 for the low (user) half). The supervisor map pointer may only be set by storing into the physical location. The user map pointer my be saved and restored by special instructions (STASK and LTASK). In addition, individual map entries in the currently active maps may be modified by the REPENT instruction.

In a multiprocessor, all processors may have their addresses mapped in the same way, or a separate map may be maintained for each processor. The Nebula architecture is carefully defined to allow a separate address space for each process (although sharing is also allowed). If all processors used the same map to resolve virtual addresses, two processors could not simultaneously run processes with different address spaces. I shall therefore assume that each processor must have an independant memory map.

Nonetheless, there remains the choice of whether mapping is done locally on each processor, with physical addresses sent to a shared memory unit, or in a central shared memory mapping unit. In the latter case, a line may be reserved to indicate to the memory unit whether a given address is to be mapped. Sharing the

memory mapping unit allows many components, such as the comparitor and ALU required for mapping to be shared among processors. However, as we shall see the mapper still has to maintain separate copies of many registers. There are other considerations, concerning concurrency and arbitration, which I will discuss later. Mixtures of these approaches are also possible. With mapping done by the processor, the memory could add yet another level of mapping, translating a "physical" address from the processor into a (truely) physical memory address. With a shared mapping unit, an extra line can be reserved to specify that an address is not to be mapped. To determine the feasibility of these alternatives, I have made a careful survey of all places in the architecture where physical addresses are used.

2.1 Physical Addresses

The Nebula architecture is designed so that memory locations are almost always specified by virtual addresses. However, physical addresses come into play in several ways: The architecture specifies some physical addresses specifically as having special meanings; it specifies that certain locations contain the physical, rather than logical addresses of other locations; it decribes some instuctions that expect physical addresses among their operands; and it partially specifies the behavior of the processor with regard to caching of translation information.

Reserved addresses

The architecture reserves the first megabyte (addresses 0-FFFFF) of physical memory for "I/O space". These addresses are to be used for "communication with I/O devices". In addition, several processor registers, such as the map pointers described earlier, the processor status, the SVC and OPEX Regsiters (which contain the virtual addresses of vector tables), and timer registers are located at assigned addresses in I/O space [p. 64].

The standard also assigns 256 addresses immediately following I/O space (100000-1000FF) to interrupt and trap verctor words, and some processor registers, such as the Software Interrupt Register, and the Kernel and Reset/IPL Save Area Pointers. I feel that this assignment is particularly unfortunate, as I will explain further below. Since many of these locations must be treated as special hardware registers rather than ordinary memory locations, I will use the term "I/O space" to include these locations.

Stored addresses

One place that the architecture requires a physical address to be stored is in a map pointer register, although the STASK and LTASK instructions copy this register to or from another location and the MAP instruction may be used to compute a physical address and store it in memory. IOC segment registers may contain physical addresses, although the standard is unclear on this point (see the discussion of SETSEG below). The map tables themselves contain values that are the difference between the physical and

virtual addresses of segments. The save area pointers (100020 and 100044) contain the physical addresses of save areas, which themselves contain physical addresses. Finally, mapping may be independantly disabled for each half of the virtual address space, in which case all addresses are interpreted as physical addresses. In particular, if the map pointer loaded by a power restore, reset, or IPL event has its low two bits clear, the restored context pointer will be treated as a physical address.

Instructions that use physical addresses

The LTASK instruction has an operand that is the (virtual) address of a two-word block of memory that is copied to physical addresses FF804 and FF828. As mentioned above, the second of these words will then be interpreted as a physical address.

The MAP instruction has an operand that is to be interpreted "in the same way that the contents of a map pointer register is interpreted" [p. 136]. Presumedly, this means that it is to be interpreted as a physical address.

The first operand of the SETSEG instruction is mapped as a virtual address, but must map to a physical address of the form XXXD0 where X denotes any (hex) digit and D is 1, 2, or 3. If certain access checks pass, the processor loads this location with "implementation dependent mapping information...sufficient to enable the IOC to distinguish valid virtual addresses with the [segment selected by the second operand] and to relocate them to the physical memory assigned to the segment, as specified by the CPU map entry" [p. 138]. It is not clear what happens if the map

entry in question changes before the IOC is called upon to map addresses within the segment. Does it use the map entry current at the time when the mapping is done, or is the mapping bound at the time the SETSEG instruction is executed? The former interpretation causes some problems: If the CPU map is changed, the portion of virtual address space specified by the second operand may no longer correspond to a unique CPU map entry (because segment boundaries have been rearranged). Moreover, the latter interpretation allows the CPU the useful option of "handing over" a segment to the IOC and then removing it from its own map (either by changing its map table or by loading a different tack with the LTASK instruction) while the IOC is performing the operation. Therefore, I will assume the latter interpretation. Under this interpretation, the "implementation dependent" information stored must be equivalent to the physical address of the segment and its bounds (as virtual addresses).

The REPENT instruction loads a map entry. Since a map entry contains the difference between a virtual and phyisical address, the source operand of the REPENT instruction should have this form.

Caching

The standard explicitly allows the mapping tables to be cached and states when the cache must be flushed (i.e., at what times the mapping tables in main memory must be made consistent with the translation cache)--namely, after any LTASK, STASK, or REPENT instruction.

2.2 Virtual Addresses to Mememory

Let us suppose that processors communicate with memory through virtual addresses. In this scheme, one memory mapping unit is shared by all processors. A virtual address from a processor is tranlated by the mapping unit to a physical address, which is then passed to the appropriate RAM bank. If the phyiscal address is in I/O space (less than 100000 hex), it is sent instead to the appropriate device.

It is possible for the mapping unit to maintain a different map for each processor. When a virtual address v arrives from processor i, the processor number (i) is latched in the mapping unit. Let us ignore, for the moment, the possibility of caching. A separate copy of the Map Pointers is maintained for each processor. The values of i and the high bit of v are used to select a Map Pointer register in the mapping unit. If this register specifies relocation and/or protection (bit 31 set), it contains the physical address of a mapping table. This physical address is used to obtain the mapping table entries from RAM to check and translate the virtual address v, obtaining a physical address p. Finally, p is sent to RAM (or the appropriate device) for the action (read or write) specified in the original request from the processor.

Suppose the processor attempts to inspect the Supervisor or User Map Pointer itself. In this case the physical address p obtained by the above calculation is FF824 or FF828, respectively. Since this address is in I/O space, it is sent to the appropriate

device, which in this case is the mapping unit itself. The mapping unit responds by selecting the correct internal register based on the value of p and the latched value of i (the processor number). Thus each processor thinks its Supervisor Map Pointer is at physical address FF824, but different processors may find different values there. Similarly, each processor may update its Map Pointers without affecting others.

Other registers, such as the PSW (FF80C) and the Auxiliary Status Register (FF820) must also be maintained as multiple copies. Registers such as timers and SVC and OPEX Vector Pointers could be shared among processors, or could be maintained as private copies, at the option of the implementation.

Now consider what happens if the mapping unit caches the map tables. In the single-processor case, the cache can be an entire map or a selection of some entries, although in the latter case, each entry must be augmented by the virtual lower bound (or length) of the segment. (A map entry normally contains only the upper bound; the lower bound is upper bound of the previous entry.) A virtual address that falls within the bounds of a cached entry can be resolved without going to memory. In the multiprocessor case, each cache entry (or table) must also contain the processor number. A cache "hit" occurs if the virtual address is in bounds and the processor number matches.

It may be worthwhile to make a special case of the instructions REPENT, STASK, and LTASK. The REPENT instruction accesses its operands in the usual manner and then sends a special command to the mapping unit specifying the map number (0 or 1), the seg-

ment number, and the 64-bit value to be placed in the map entry. The STASK instuction sends an address, which is translated to a physical address p in the usual manner, and a special signal that directs the memory unit to send the User Map Pointer corresponding to the sending processor (as well as the Task Context Pointer) to RAM at address p. The LTASK instruction is treated similarly.

If a central mapping unit caches translation tables but does not have special commands for REPENT, STASK, and LTASK, it must understand a signal that directs it to flush the translation cache corresponding to a given processor.

Note that under the scheme just described, the processor always communicates with memory through virtual addresses sent to the mapping unit. The processor never generates physical addresses.

2.3 Physical Addresses to Memory

In the above scheme, one mapping unit is shared among all processors, but it generally treats each processor separately. An alternative is to associate a separate mapping unit with each processor. The translation algorithm is almost exactly the same. The only difference is that the processor number need not be latched on each reference, but can be "wired in".

2.4 Interrupts and Traps

When an interrupt or trap occurs, the processor goes through roughly the same sequence of operations that occur on a procedure call, execept that the entry point of the "procedure" is taken from a specified location whose physical address depends on the nature of the trap or the device that is interrupting. For I/O devices, the use of a physical address is not particularly onerous, since the same routine will probably be used to service an interrupt from any given device, regardless of which processor executes it. However, this does represent the only case in which a processor must bypass memory mapping and send a physical address directly to memory. There is also the question of which processor gets the interrupt. I suspect that in most applications, the binding of I/O device to CPU will be fixed. Nonetheless, the architecture only leaves room for interrupt vectors for 10 distinct devices. Either we are restricted to at most 10 devices regardless of the number of processors, or these locations must be maintained in multiple copies, much as the Map Pointer of PSW is maintained in the scheme outlined above.

For processor interrupts and traps, the problem is worse. For example, an invalid memory access causes a trap to the procedure whose address is stored in physical location 10000C. Since different processors may be executing quite different processes when such a trap occurs, it is likely that they will want to establish different trap handlers. There are two solutions: We may resign ourselves to having one routine for all

memory management traps. Since the Context Pointers are (potentially) different for different processors, the routine could tailor its actions according to values ultimately accessed via these pointers. This solution seems to abandon many of the advantages that the exception-handling mechanism of the Nebula architecture provides. The other alternative is to maintain multiple copies of physical memory locations such as 10000C.

Some of the "Interrupt Vectors" are not interrupt vectors at all. For example, the Software Interrupt Request Register (100004) has the side effect, that storing into it can cause an interrupt. It is clear that this register cannot be implemented in RAM. It is also clear that there must be a separate copy of this location for each processor, or else there would be no way of knowing which processor to interrupt when the location had its contents altered.

All of these considerations imply that locations 100000 to 1000FF of physical memory must be treated very specially. I have no idea why these particular locations were not mapped into I/O space or why only 256 bytes were dedicated to vectors, whereas one million bytes were dedicated to device registers. I strongly recommend that these assigned locations be moved to I/O space. Otherwise the memory controller hardware is significantly complicated: Instead of simply testing whether the high order 12 bits are all zero (in fact, the standard allows physical memory space to be limited to 24 bits, so that only the high 4 bits need be inspected), the controller must do a comparison to determine whether to treat an address specially.

2.5 Other Considerations

Interprocessor Control Under this scheme, there is no way for one processor to control another by modifying its map pointer without extending the architecture. However, it could modify the map table itself using the REPENT instruction, since the REPENT instruction uses a map pointer operand rather than the processor map pointers. In the presence of caching, their are two alternatives: A REPENT by one processor could cause a flush of all affected caches (or more likely, all caches, since it is not so easy to tell which ones might be affected), or a REPENT by one processor that affects a map table of another processor might be declared "undefined" and avoided by software.

Connections to Memory and Other Devices If an address maps to I/O space, the mapping unit must dispatch the request to RAM, a CPU, an I/O device, or itself, depending on the specific address. If processors communicate with the mapper using a shared bus, some addresses must be extended by a processor number, since multiple copies of an address may exist.

Arbitration Since each processor has its virtual addresses resolved independantly, multiple processors can do the address translation concurrently. If the resulting physical address p selects RAM, it is sent to RAM over a shared bus or directly to a multiported RAM bank, where competing requests from different mapping units are arbitrated. These requests may be translated requests from the CPU's or they may be requests from the mapping

units themselves, for example for mapping table entries. Similarly, some scheme is required to arbitrate amound competing requests to an I/O device. If a virtual address from the CPU translates to one of the Map Pointers, it can be handled directly in the mapping unit (although it may generate a cache flush that requires access to RAM).

3. MULTICOMPUTERS

The suitability of the Nebula architecture for use in multicomputers is almost entirely determined by the IOC structure. Unfortunately, this section of the standard is the most poorly written. The IOC seems to have been designed with existing hardware peripherals in mind, but very little hint is given what these peripherals look like. In particular, there are numerous references to "1553B" but no cross-reference to a place where the reader can find out what "1553B" is. There are also cryptic references to "serial point to point" and "parallel point to point" channels. The instruction set for the IOC is surprisingly elaborate. There are features whose purpose I can only guess at. Once again, I suspect that some of these features are present to accomodate uspecified existing hardware.

My experience on the Arachne (formerly Roscoe) project [6] has taught me that the most important feature of a processor for a multicomputer (other than those features, such as memory management, important in any general-purpose computer) is a good front-end processor that can handle communications. At the very

least, the front-end should be able to carry out the low-level details of whatever communications protocol is in use concurrently with the main processor. The IOC design is missing two important features required to do the low-level protocols: The less important omission is bit-manipulation instructions to compute checksums and perform bit-stuffing. There are logical operations and shifts, but they are not sufficiently special-purpose to do the kind of manipulations required, for example, for HDLC in a reasonable amount of time. On the other hand, these operations would most profitably be offloaded to the communications device itself. (There already exist chips that do the bit-stuffing and CRC calculations required for HDLC).

The more serious omission is some sort of timer support. All reliable protocols I know of require some sort of timeout. Since the IOC would be doing nothing but controlling communications, the timer support could be extremely primitive: a simple time-of-day clock would suffice. The ideal support would be the same kind of interval time as is provided to the main CPU: a register that is decremented periodically and interrupts the IOC when the count drops to zero. This kind of timer could be simulated using a time-of-day counter as follows: To schedule an "interrupt" n "ticks" in the future, the processor could read the clock, add n, and store the result in a local variable, say "next_event". The processor would then perform its other duties on a "round-robin" basis, periodically comparing the clock against "next_event". When the clock passes "next_event", the processor would schedule the task associated with the timeout

event. Communications protocols are typically characterized by a requiring, at any one time, that a small set of simple (and fixed-length) tasks be performed.

Unfortunately, the standard associates the time-of-day clock with a fixed physical address. The only way that the IOC could gain access to the clock would be for it to be mapped into the Program, Message, or Data Buffer segment. However, the standard prohibits mapping addresses less than 100000 into these segments (section 13.4, p. 48), and the timer is at location FF840 in I/O space. I therefore strongly recommend that some sort of time-of-day clock (at the very least) be made accessible to the IOC.

By the way, the standard specifies that the time-of-day clock be incremented every 10 msec, which seems to be a rather strange value. A resolution of at least one millisecond would seem to be crucial for many applications. I suspect that many implementations would ignore the time-of-day clock entirely and use one of the interval timers (which have microsecond resolution). The choice of 10 msec is sufficiently close to line frequency to suggest that the designers intend the time-of-day clock to be driven off the line current. But in the United States, line current is 60 Hz, which gives a tick every 16.67 msec, and in most of the rest of the world, it is 50 Hz, which works out to a tick every 20 msec. I would suggest either following the approach of the IBM 360 and increment the clock either by five 60 times per second or by six 50 times per second (giving an effective rate, but not resolution, of 300 Hz), or have one tick each millisecond.

Given that the processor clock is not visible to the IOC, two alternatives are possible. One is to put the timer support in the communications device itself. The IOC could read the timer by executing a READ or READS command. This solution is clumsy and seems to be out of the spirit of the standard. (If functions are to be pushed out to the device itself, what is the point of specifying the instruction set of the IOC in such detail?) The other solution is to require more of the protocol work to be done by the main processor.

If the processor is to do any of the protocol itself, a "scatter-gather" facility would be very helpful. That is the ability to specify a sequence of buffers in a single I/O command. On output, the buffers are gathered together and sent as one packet; in input, the packet is broken into pieces and scattered to the different buffers. This feature is usuful to obviate the need to copy a message. The message could be presented directly as one buffer and any protocol headers could be in other buffers. The IOC standard provides this feature in two ways: First, the address of the buffer in a READ or WRITE operation is specified by a word in the message. Therefore, the message can contain the addresses of a whole list of buffers. Second, the messages themselves can be chained. Unfortunately, all of these buffer addresses are virtual addresses which are mapped according to a single segment specified by the Data Buffer Segment Specifier. This means that all the buffers in a single I/O operation (from the point of view of the main processor) must lie in a single contiguous block of physical memory, all of which is mapped to

the IOC for the duration of the operation. However, the situation is not hopeless. The protocol headers could be put into the message itself and read/written with the RTMSG and WFMSG commands. This at least allows the headers and bodies to come from different portions of physical memory. However, this scheme may prove insufficient for multi-layered protocols. On the whole, I think communication with the IOC through physical addresses would make much more sense.

Another feature that would be useful for a multicomputer, especially one built using a broadcast medium such as Ethernet or a token ring, would be a sophisticated message-screening facility in the front-end. Ideally, the main processor should be able to present the front-end with a set of commands of the following form: If a packet arrives with such-and-such characteristics (based on the sender and on some sort of "subject" field in the packet), then take the following actions (which could include returning some sort of canned response, putting the packet in a particular place depending on its header, and/or interrupting the main processor). The "message chaning" facility of the IOC seems to be of little use here, since all the requests must remain active simultaneously and there is no way for the IOC to go back to a message once it has chained to the next one. However a set of such requests could be constructed by building an appropriate channel program and a corresponding message. Once again, though, the restriction that the message can only point to locations in a single segment limits the usefullness of this approach. It would not be possible, to give a list of buffers in different user

processes and instruct the IOC to put a packet into one of them based on its header.

4. MISC OBSERVATIONS

Reading the standard I came upon a few errors that, while not specifically related to the subject of this report, might be of interest to the framers of the specification:

Most importantly, in the discussion of memory mapping (p. 42) it is not clear whether the "relocation amount" is to be interpreted as a signed quantity (it almost certainly must be) and if so, exactly how sign-extension is to be performed.

The description of the MAP command (page 136) states that Ptr is interpreted "in the same way" as the contents of a map pointer register. It is not clear whether this implies a physical address.

In the description of the REPENT instruction (p. 135) there are two typographical errors: Under "Operand types", the second line ("16-bit, 32-bit, 64-bit logical") should read "A 16-bit, 32-bit, 64-bit logical". The word "an" in the last line on the page should be "and". It should also be more carefully specified how A is to be extended to 64 bits (since a map entry must be 64 bits long).

On pages 37-39, all physical addresses of interrupt and trap vectors are off by 100000 (hex).

On page 37 there is the cryptic notation "(see section ref[iocint])", which probably should read "(see section 13.7.1)".

The description of the SETSEG instruction specifies that the "Access Code Required" for the Channel Program segment specifier must be "Instuction". There is no such access code. The code should be either "Execute" or "Execute/Read". It is not clear which is intended.

5. REFERENCES

[1] Military Standard, "Nebula instruction set architecture (draft)," MIL-STD-1862A (May 28, 1980).

[2] R. J. Swan, S. H. Fuller, and D. P. Siewiorek, "Cm* -- a modular multi-processor," Proc. National Computer Conference 46, AFIPS Press, pp. 637-644 (1977).

[3] R. J. Swan, "The Switching Structure and Addressing Architecture of an Extensible Multiprocessor: Cm*," Technical Report CMU-CS-78-138, Department of Computer Science, Carnegie-Mellon University (August 1978) PhD Thesis.

[4] R. J. Swan, A. Bechtolsheim, K-W. Lai, and J. K. Ousterhout, "The implementation of the Cm* multi-microprocessor," Proc. National Computer Conference 46, AFIPS Press, pp. 645-655 (1977).

[5] A. K. Jones, R. J. Jr. Chansler, I. Durham, P. Feiler, and K. Schwans, "Software management of Cm* -- a distributed multiprocessor," Proc. National Computer Conference 46, AFIPS Press, pp. 657-663 (1977).

[6] M. H. Solomon and R. A. Finkel, "The Roscoe distributed operating system," Proc. 7th Symposium on Operating Systems Principles, pp. 108-114 (December 1979).

Miscellaneous Problems with the Nebula Architecture

Jim Elkins

Robert Cowles

Digicomp Research Corporation

Ithaca, NY 14850

References to Nebula in this report refer to the version of

MIL-STD-1862A dated "TBD"

ABSTRACT

A number of problem areas with the Nebula Instruction Set Architecture (MIL-STD-1862A) are discussed which were not covered in other reports by the independent reviewers, but which Digicomp felt deserved some emphasis. This report is based on the version of MIL-STD-1862A issued in late September, 1982, dated TBD.

INTRODUCTION

This report covers miscellaneous problems and suggested changes not in the domain of the other reports. It is organized into four sections. The first discusses problems related to exceptions, interrupts and traps. The second raises terminology problems and points needing clarification. It often deals with situations which are not covered in MIL-STD-1862A, but which can arise with legal implementations of the standard. The third section contains comments about various portions of the architecture which were considered problematic but for various reasons were not studied in detail. The fourth section is a discussion of the goals of the Nebula program versus its achievement of those goals.

TRAPS, INTERRUPTS, EXCEPTIONS

These are discussed together because they share some problems. Although Nebula closely matches Ada's rules for exception handling, there are several problems with exceptions in Nebula.

1. Nebula does not specify (precisely) when an exception or trap can be raised during execution of an instruction. This omission might be especially troublesome in pipelining where Nebula leaves much freedom to the implementor. An implementation might allow an instruction to raise an exception during operand pre-evaluation while the previous instruction is still executing. It is legal in

Nebula that an exception could be discovered and raised during pre- evaluation of operands of the next instruction in the instruction stream even though the current instruction has not finished executing.

2. Truncation on integer operations completes the instruction and writes to the result before raising the truncation exception. This is inconsistent with Ada, which requires that the result operand not be changed. Ada compilers on Nebula will therefore have to use a temporary for the result unless analysis can determine that an overflow is impossible. Integer truncation is also inconsistent with the rest of Nebula in two ways.

   * Overflows on floating point operations do not write the result if the exception is enabled.
   * On all other exceptions in Nebula the instruction is aborted with no side effects.

   Nebula should be changed such that, if the result of integer arithmetic instructions can not be correctly represented in the destination and the EAE bit is set, then the truncation bit should be set, the instruction aborted with no side effects, and the truncation exception raised.

3. There are three problems with the instructions which raise exceptions.

* Since exception codes of 0 are allowed, if the ECODE instruction returns a value of 0 then the program will not be able to determine whether the exception handler was in the "exception-code-available" state with an exception code 0 or the exception handler was in the "disabled" or "handler defined" states.
* Since instructions can raise exceptions with exception codes of 1 through 35, the exception handlers cannot determine if an exception was raised by the hardware or the software.
* No exception codes are reserved for future use by the hardware.

The latter problem is particularly easy to resolve at this point, but may be very costly if resolution is delayed until a significant quantity of software is developed. The problem should be resolved.

4. Some instructions in Nebula are required to be interruptible, for instance Block Move and Compare Block. All instructions are potentially interruptible; whether they are or not is implementation dependent. The operating system, in handling traps and exceptions, may need to know if the instruction has been completed, suppressed or interrupted. If interruption is the case, the operating system might need to know something about the instruc-

tion's suspended state to process the exception. Neither sort of information is presently specified.

5. Under some conditions the supervisor exception handler (SEH) will not be able to tell if it is returning to the environment of the exception or to the environment of a direct or indirect caller of environment where the exception occurred. For example, if, on an exception, the UDLE bit is cleared and the exception handler is disabled, the exception will propogate up the call chain. If the caller has its UDLE bit set a supervisor exception handler call will occur. The way the SEH determines whether the exception occurred in the procedure context which invoked the SEH is by comparing its second and third parameters. If they are not the same then the exception must have been propagated. (This assumes particular answers to other questions concerning exceptions.) However if parameters 2 and 3 are the same one cannot conclude that the exception occurred in the procedure context that invoked the SEH. If the exception occurred in the instruction following a recursive call of the procedure, then parameters 2 and 3 would be equal whether the calling procedure or the called procedure invoked the SEH.

TERMINOLOGY AND POINTS NEEDING CLARIFICATION

The standard needs to be rewritten for clarification and improved precision of terminology and explanations. While the standard has generally improved in these respects since the 1980 May 28 version, there are still many situations which can arise in Nebula implementations with the ability to affect the software but which are not mentioned in the standard. Reviewers have frequently misunderstood points in the documents and have had to ask numerous questions for clarification. Too often, one must be familiar with the rationale of the designers to answer questions and clarify ambiguous or incompletely specified points.

This section is meant as a partial list and summary of some of these issues. It is not definitive and generally does not recapitulate issues raised by other reviewers, whether or not Digicomp feels they were addressed satisfactorily. Of course, page and section references refer to the version of MIL-STD-1862A issued in late September, 1982, with "date TBD".

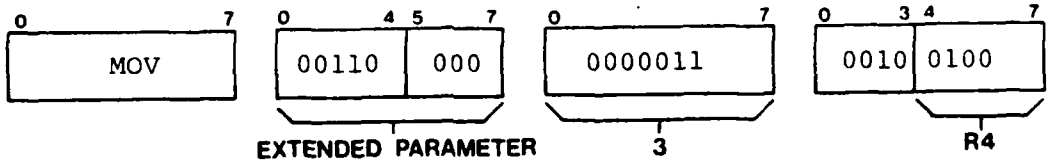
General Points.

1. The definition of "interruptability" (page 3), "An instruction is restartable from the point of interruption", needs some clarification. What action can cause an instruction to be interrupted? Is it restricted to interrupts or can it be an exception or trap? Can an interruptible instruction abort under some conditions instead of interrupting and resuming from the point of interruption?

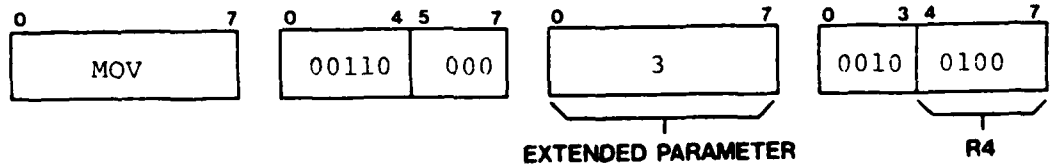
2. When an exception occurs the presently executing instruction can terminate, abort with side effects or abort with no side effects.

3. "Interruptable vector instructions" (page 69) is undefined and should be replaced with "interruptible instructions".

4. The last line of the example on page 14 should read



rather than:



5. In section 11.3 on page 40 "physical address 4" should read "physical address 100004".

6. In section 11.4 on page 40 and in section 11.5 on page 41 where "word at physical address" is used, "vector at physical address" is probably intended.

7. In paragraph 5, page 41 "in step 2 used" should read "in step 2 is used".

8. The last paragraph of section 11.6 (page 41) states that

the ability to detect hard memory errors is implementation dependent. However the last sentence on page 45 states that "a hard memory error is generated". It is not clear in this case whether the processor can generate the error but not take the trap or the processor must trap through location 100014 (Hex).

9. In section 11.11 on page 42 "simultaneous occurrence" needs to be explained.

10. In Chapter 30 it states that "opcodes reserved for hardware implementors shall produce implementation dependent unpredictable results". Is the word "unpredictable" intended? Are all or any of these opcodes used for privileged instructions? If some of the opcodes can be privileged, who makes the determination?

11. If the operand specifier for the index in the scaled or unscaled index mode has a size of 64 bits then the address is undefined (Sections 5.9 and 5.10, pages 16, 17). This seems to be an unnecessary implementation dependency. An exception should be raised.

12. The explanation of the last mode bit of the PSW (section 6.2) should be rewritten. This bit indicates on which context stack to resume operation when returning from a task, i.e from a procedure context with the base bit, PSW bit 16, set. This bit is set appropriately on procedure

invocations which are task initiations, i.e. are caused by interrupts, traps, TINITs or PINITs; otherwise it is not changed.

13. The second sentence of section 6.14 should begin with "In instructions which are not floating point operations".

The Procedure Interface.

The procedure interface is one of the fundamental components in the design of Nebula. Documentation of the interface is generally well written and comprehensive. However, slightly more detail is needed and several points need clarification.

1. OPEXs and Supervisor Exception Handler calls should be added to the list of mechanisms that invoke procedures in section 4.3 (page 4).
2. Chapter 8 explains the procedure interface and all the methods for procedure invocation. It does not, however, explain the process of returning from a procedure in detail. This explanation should be added.
3. In section 8.1.2 point 4, "the state of the exception handler" should be changed to "the state and the address, if any, of the exception handler". The procedure context contains more than just the state of the exception handler.
4. The terms "context stack", "active context stack" and

"context area" need to be defined. The first term is used frequently. The latter two are used in section 8.1.3, which states "the representation of the context area of the active (Kernel and Task) context stacks is IMPLEMENTATION DEPENDENT". This phrase has been given many interpretations. A "context stack" contains a collection of one or more execution contexts. However, it is not specified what marks the boundaries of the context stack. For example, if protection is on, is the whole context stack always marked "context access only"? Also, "active context stack" needs to be defined before one can talk about dynamic management of the "context stack".

5. Section 8.1.5, Alignment of Context Pointers, needs clarification. Must the context pointer point to an object in the current procedure context? Are the references to "context" in this section references to a "procedure context"? The last sentence states that a context area may be initialized by setting the context pointer to the greatest word address in the context area plus 4". Again, what is a "context area"? Is this a statement for programmers or implementors, i.e can a programmer initialize a context area by setting the context pointer to this value or is the value to which the context pointer needs to be set for initialization implementation depen-

dent? It would seem to be the latter. The former case would seem the appropriate one to define; even if the context area is implementation dependent, there should be an implementation independent way to initialize it.

6. It is not clear what will happen if a program has two "context access only" segments which are contiguous in virtual space but discontiguous in physical space. The standard does not specify how the procedure interface must handle this situation. Thus, a program cannot be dynamically allocated context stack space as needed.

7. It should state in the third bullet of section 8.3 that the exception handler is put in the disabled state.

8. In section 8.3.1, under specifications for when the BASE bit (PSW:16) is set, invocation of a supervisor exception handler with a Task.Failure exception should be added.

9. In section 8.4, Parameter Lists, it states that "in certain interrupts and traps [the parameter list] is implied by the architecture". This is the case in most, if not all, interrupts and traps. The term "certain" should be deleted and any exceptions to this should be listed. Also the supervisor exception handler should be added to the list of procedure invocations with implied parameter lists.

10. In fig 8.6, "double integer" should be deleted.

Exceptions

1. Figure 9.1 shows the three states of the exception handler and some of transition paths between them. When it was pointed out that this did not show all the allowed transitions, the text was changed from "the allowed transitions" to "the typical transitions". It would be much more useful to have a diagram which showed the allowed transitions. Figure 1 below is an attempt at such a diagram.
2. Figure 9.2 shows the mechanism for propagation of exceptions. However the box labelled "START" is really the start if the exception is a hardware exception or was raised with RAISE. It would be useful if the diagram included exceptions raised with ERP, ERET, TRAISE and PRAISE. Figure 2 below has been modified to include these exception sources.

Points related to the IOC:

1. In specifying the BIT bit (bit 15) of the Channel Status Register (section 13.2.4), Nebula should specify that a "0" denotes no error detected and a "1" denotes the detection of an error.
2. "Channel dependent" is used in Chapter 13. This term should be defined. For instance, since channels are standardized but different, it could mean that the field

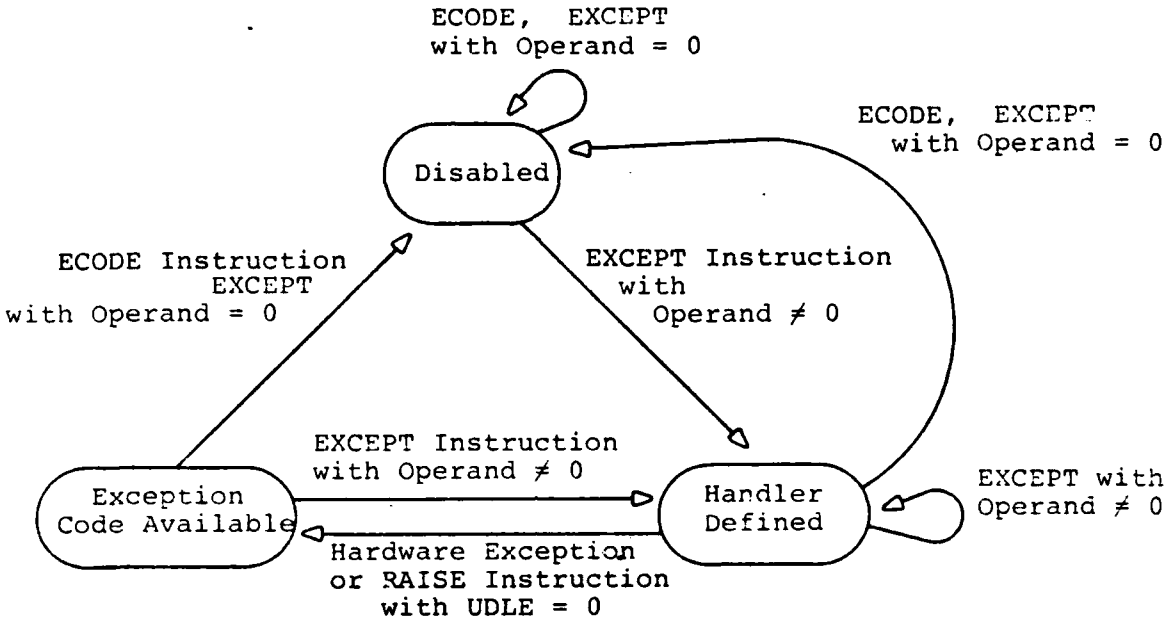


Figure 1: Allowed State Transitions During Exceptions

means one thing for all 1553B channels but possibly something else for SPP channels. This seem to be the meaning when it is used in paragraphs 1 and 2 of section 13.8.1. It is not necessarily the meaning one would assume in its use in describing the Channel Configuration Register and

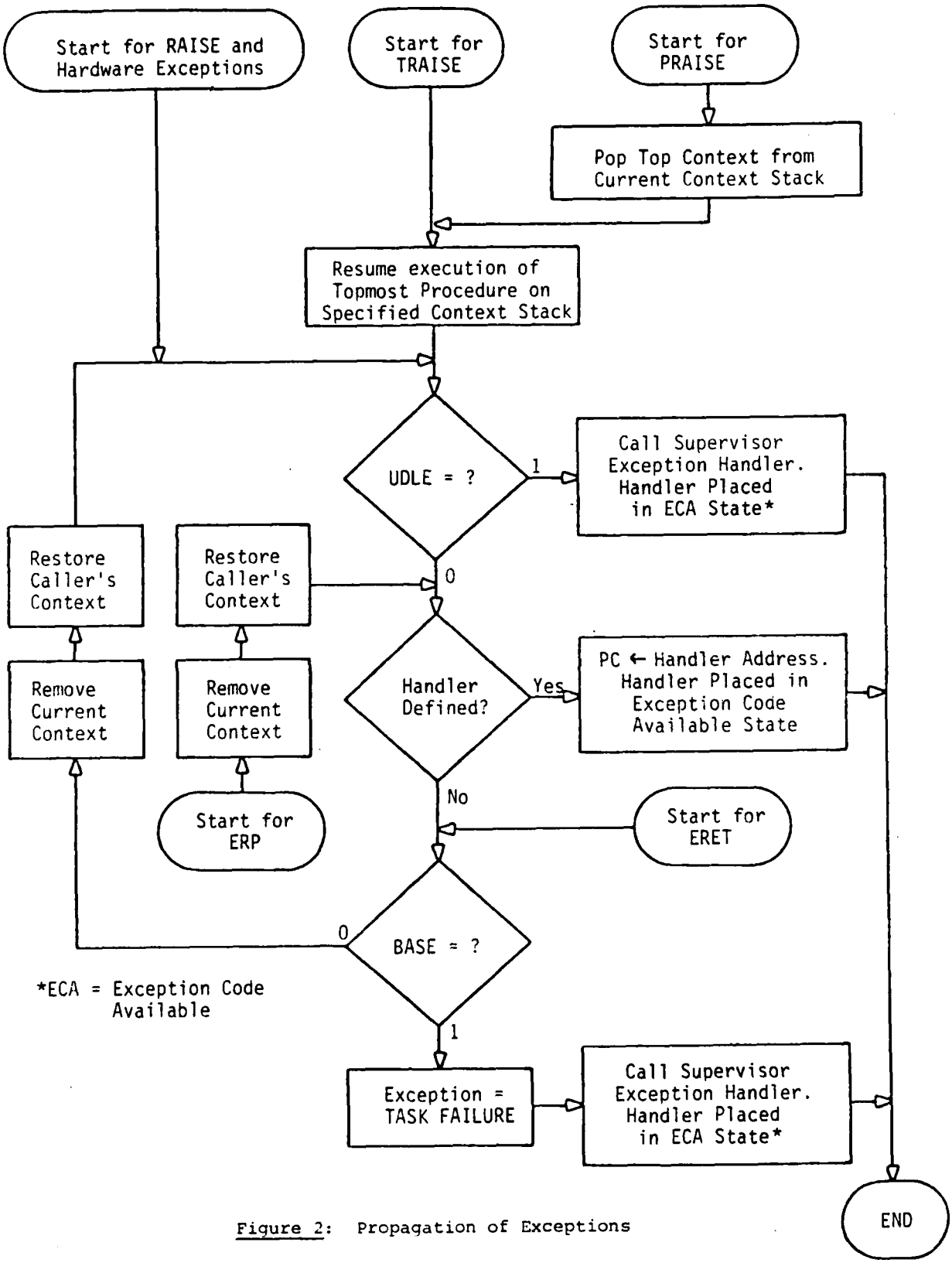


Figure 2: Propagation of Exceptions

Channel Status Register. The former definition would significantly aid portability of channel programs.

3. Section 13.8.4.2 mentions the input and output channels of the SPP. These should read "channel program" rather than "channel".

4. The term "implementation reserved" used in describing the IOC register blocks on pages 67 and 68 (section 13.9) should either be defined or changed to a defined term.

5. If relocation and protection are off, how does the SETSEG instruction work?

Points related to the instructions

1. In the instruction descriptions (chapters 17-29) one field of the description lists the exceptions and traps that can occur in executing the instruction. Not all possible exceptions and traps are listed since many are not instruction specific, i.e., either they can occur in all (or most) instructions or they can occur in none. It would be helpful to list the subset of the exceptions and traps that are never mentioned in the program exceptions section of the instruction description. An alternative would be to list the subset in the explanation of the instructions descriptions in Chapter 16.

2. Operands are specified by either operand specifiers, in-

line literals or displacements. Displacements should be listed as a type in section 16.2 and discussed in a subsection. Also the first sentences of sections 4.2 and 5 mention neither in-line literals nor displacements in describing instruction formats.

3. There is an inconsistency in the setting of the N bit in the subtract and compare instructions. A SUB #3,#4,%4 instruction will clear the N bit, while a CMP #3,#4 will set this bit. This is also true of the SUBU and CMPU instructions. Nebula should be changed so that the N bit in CMP (& CMPU) is set when "B lss(u) A" rather then when "A lss(u) B".

4. It is not clear exactly what is allowed in implementing the Compare and Swap instruction (CMPS). CPMS is an interlocked instruction which requires write access to its second operand if the values of the second and third operands are equal and to its third operand if they are not equal. Is it legal to require read-write access for either the second or third operand before starting the instruction and initializing the interlock on the second operand. Depending on how this is handled there are circumstances where a memory management trap will or will not occur during execution of the instruction. Only one of these interpretations should be allowed.

5. In the CALL, CALLU and SVC instruction descriptions, the

operand type for the parameters, P1,...,Pn, is listed as "all types of operands allowed in parameter list". The standard should either list the types allowed or explain this phrase in the description section.

6. In the LTASK instruction the second word in the two-word block contains the physical address of the map. It should contain the status bits of the map pointer register as well, i.e. this word should have the same format as a map pointer register.

7. A specification.error exception can be raised by the load task instruction (LTASK) (See section 12.2.1). LTASK should list this exception under its program exceptions section and describe the condition which causes it to occur.

8. The action of REPENT under exceptional conditions is not clear. What happens if the segment number is larger than the current map size or if it is larger than the maximum map size for that implementation?

9. The use of the Z bit in MAP and SETSEG is somewhat inconsistent. In MAP, the Z bit is cleared to indicate an improper virtual address. In SETSEG, the Z bit is set to indicate an improper virtual address. MAP should invert its use of the Z bit to remove this minor but needless inconsistency.

10. The operation of the Wait for Interrupt instruction

(WAIT) is unclear. The description states: "The processor shall enter a wait state until any interrupt is posted, at which point it shall respond normally." "Posted" needs to be defined. If it means "accepted" then the processor will remain in a wait state until a interrupt of higher priority than the current task occurs. This interrupt will be processed and then the instruction following the wait will be executed. If "posted" means "pending" then the processor will complete the wait instruction and execute the next instruction as soon as any device is trying to interrupt the processor. The processor will not know, however, whether or not it has processed an interrupt between the start of the WAIT and the execution of the instruction following the WAIT.

GENERAL COMMENTS ON NEBULA

This section contains some general comments on Nebula. The issues mentioned are not necessarily problems with the architecture, but bear mentioning.

Memory Management System

The memory management system is probably the area of Nebula that is most often misunderstood and, aside from the procedure interface, been the most commented on. It is also unclear what the requirements are for

this area; thus making analysis of its suitability difficult. The following are general comments on it:

1. The Memory Management System does not support "demand paging", that is, if an instruction causes a memory management trap because it referenced non-resident data, then the trap handler is not able to fetch the missing information from secondary storage and transparently resume execution at the point of the trap. Demand paging is made difficult by several parts of the ISA and is ruled out in the introduction to Chapter 26 which forbids alteration of the source or destination region of a string instruction after the instruction processing has begun.
2. On a memory management trap, the ISA restricts what the trap handler can do. Assuming the program counter is pointing to the instruction causing the trap, then it is hard to skip an instruction; if the program counter is pointing to the next instruction then it is hard to execute the instruction which caused the trap; finally if what the program counter points to is implementation dependent, then the trap handler does not know what will happen if it executes a RETurn to the task causing the trap.
3. The map entries describing a segment are interdependent.

The segment is described by two entries, one providing the upper bound and the other providing the lower bound. The upper bound of a segment is the lower bound of the following segment. This design implies that it is believed that usually the virtual address space of a program will be contiguous. In places where the virtual address space is noncontiguous, a map entry must be used to mark the nonused space. Since there are a limited number of segments, to avoid wasting them to mark unused virtual addresses, the program must use contiguous virtual addresses. If it is necessary for a program to, as a rule, use noncontiguous segments, then that program will only have half the actual map entries available for segments.

4. Aliasing of physical adresses (having distinct virtual addresses map to the same physical address) leads to unpredictable behavior (Section 12.3.2). This means that the supervisor and a user cannot both have access (perhaps with different access rights) to the same code.

5. If address relocation is needed in a program, then protection is required to be used.

6. SETSEG must use one of the active maps. If I/O is being done into a user space, then requiring that user's map to be active may be inefficient. If there is an active

queue of programs awaiting access to the controller, then the new program being granted access to the IOC will not be executing. Thus to start the I/O the present user's map will have to be swapped out so that the required map can be loaded and a SETSEG executed. This map will then have to be swapped out and the original reloaded.

Comments on Procedure Interface

There has been considerable comment on and problems raised with the procedure interface in the various reports. This section mentions some points not raised before.

1. All registers used by a procedure are saved when that procedure executes a call-type instruction. Other than Register 1, no registers are inherited. This may be somewhat space inefficient since unnecessary registers will be saved in memory. Lack of register inheritance also interfers with passing information between procedures; system state information (e.g. a pointer to the activation record of a lexical parent) must be passed either through parameters or memory. To be used, however, this pointer will probably have to be loaded back into a register since the architecture has no useful pro-

visions for indirect addressing through memory.1

2. The entry address of a procedure is required to be on a double word boundary. If the operand of the call instruction is not double word alligned, then the two low order bits are forced to zero, no exception is raised, and the procedure call is made.

3. Immediately after a procedure call, the registers are undefined. Because of security and portability considerations, the registers should be either defined as a specific value or else register reads should not be permitted unless the register has been written, i.e. an uninitialized register exception or trap should occur if illegal register reads are attempted.2

4. Nebula's treatment of the PSW is somewhat unusual.

---

1 There is a way to address indirect through memory but it is a very long operand specifier (typically 7-11 bytes) and will probably not be used in favor of a two instruction sequence which will be two bytes shorter and perhaps faster.

Using special implementation techniques, the time inefficiencies of saving and restoring the registers can be greatly reduced. However, with typical treatment by the implementors, saving and restoring the registers would be time inefficent; therefore, a different implementation strategy for registers is indicated by the ISA. The strategy may prove to be a burden on microprocessor or low end implementations of the ISA, with the result that in these implementations the register treatment may be more inefficient than the more standard register architecture in other ISA's. The efficiency of the register architecture is very related to implementation strategies; however, a potential problem does exist.

2 This was agreed to be a problem at a Tiger Team meeting, but apparently the standard was not modified to correct it.

* The PSW of the executing procedure is not directly accessible. The only allowed method to read or write the entire PSW is to use the LPSW and SPSW instructions. These instructions load and store the PSW of the calling procedure.

* There is no way to read the PSW of the top procedure context of the inactive stack (actually of the top procedure context of any execution context). Thus the memory management trap handler cannot read the PSW of the procedure which caused the trap.

* The unprivileged user has very limited access to any fields of the PSW.3 The only direct way to read or write these fields is using the Set Condition Codes (SETCC) instruction which allows the user to set or reset five bits (EAE,C,T,N,Z) of the PSW. It does not allow the user to set some bits and leave others unchanged. It will typically be difficult for an unpriviledged user to read these bits.

5. When allocating a procedure context during procedure invocation, space must be provided on the context stack for temporary storage of the state of interrupted instructions. Otherwise, a context stack overflow and hence a memory management trap could itself occur when

---

3 The LPSW and SPSW instructions mentioned above are privileged instructions.

processing an interrupt or trap, and there would be no room in the stack for processing this second trap.

6. When a new procedure context is added to the current context stack, the space for saving state in the previously current procedure context should not be retained unless the space is being used. This will only happen if the new procedure context is being allocated as a result of interrupting an instruction. As a consequence, when returning from a procedure context, the implementation needs a method of knowing whether the procedure context being returned to is one with state save space. The problem is especially complex for call instruction (CALL, CALLU, SVC). These should be interruptable since they may have many parameters and therefore take too long to ensure adequate interrupt response. (Aborting calls rather than interrupting is probably undesirable.) However, the save area for the procedure context of the calling instruction must be retained until the call instruction is completed, and then and only then it should disappear.

GOALS VERSUS ACHIEVEMENTS

Introduction

In analysing Nebula one should look at the relevant goals, uses, and environments. Other reports have been primarily concerned with types of uses of implementations of the ISA. In this chapter, both the relationship between goals and environments, and the achievement of the goals are examined. The first section discusses the relevant differences between the environment and goals of the Army and Air Force and the effect of these differences on evaluation of Nebula. The second section discusses the goals of Nebula and the Air Force High Level System Standardization program and the extent to which these goals have been attained. The final section discusses the general program development and test limitations due to lack of complete specification of the hardware operations in the Nebula standard.

Army vs. Air Force Goals and Criteria

The differences between the point of view of the Army and that of the Air Force can lead to significantly different technical evaluations of the architecture's suitability. An obvious corollary to the theory of standardization is that standardization should be across all three services. Although there are cases where it has been successfully argued that a standard of one service is not appropriate for another, it should be quite possible to design an ISA that would be appropriate for both the Army and the Air Force. Inter-service differences are not based so much on differences in the embedded application as they are on differ-

ences of situation or environment. These differences and some implications of them on the Nebula effort are discussed below.

Multiple vs. Single vendors: The Army plans to purchase implementations of production versions of the ISA from a single vendor over a set ordering period. These ordering periods are intended to be a maximum of 5 years, with the first being 5 years. The Army will then supply the computers as GFE for systems needing embedded computer capabilities. Essentially, by using a single vendor the Army has temporarily avoided the problems of portability due to lack of clarity or completeness in the Nebula standard (this assumes a contractor will make the mini- and micro-computer versions of Nebula such that programs are portable between them). The Air Force will have to adopt a different policy toward the Nebula standard if they are going to allow acquisition of ISA-validated embedded computers from any vendor. To insure portability, the Air Force will have to deal with the implementation dependencies, and to eliminate areas where the standard is incomplete.[4] In general, the Air Force can be expected to have to face numerous portability problems.

However, the Army's acquisition policy is only a short term solution to the problems induced by the current standard. In order to handle portability problems that will arise during the first technology insertion stage, the Army will have to address the fact that the "standard"

---

[4] These changes to the standard would have to be handled like the Army's specification in the MCF Advanced Development contract for the Initial Program Load sequence.

is largely defined by the implementation they have purchased. That "standard" includes much more than is defined by MIL-STD-1862A. If the micro-computer and mini-computer versions are not compatible, then the Army will immediately be faced with portability problems. Given the freedom in pipelining and caching that the Nebula specification provides, this incompatibility is likely. Further development or refinement of future ISA's will have to take into account the portability of programs that were written for the "real" standard even though it may be difficult to document. Also, based on current schedules, work is to start on the future implementations before there is significant field experience with using Nebula in deployed systems.

Mainframes: The Army has expressed disinterest while the Air Force has expressed mild interest in mainframe (or large general-purpose mini-computer) implementations of the ISA. While this may not turn out to be an Air Force need, it should be addressed. The suitability of Nebula for implementation on a mainframe was not specifically studied. However, MIL-STD-1862A is very likely to prove inadequate for this. Certainly, the memory management system of a mainframe computer has to allow for many more than 16 segments. A workable scheme to allow paging (with appropriate additions to the architecture like reference bits) is also a necessity. Further, the importance of virtualizability of the ISA would likely increase.

Continuous vs. Discrete Technology Insertion: The Air Force, through competition between suppliers, has the opportunity to have continuous technology insertion. The Army plans a technology insertion point at approximately five year intervals.

The Air Force is in a position to take advantage of major improvements in implementation technology (how the hardware is built). By having continuous technology insertion, the Air Force will likely have a much better implementation in the field during the period of time that the Army's next version of the Nebula computer is being developed and readied for production. However, since the Air Force must take care not to severely impact the larger number of suppliers and contractors that are building Nebula computers, they may only make minor changes to the Nebula standard. This restriction means the Air Force must be very confident that the adopted Nebula standard meets their requirements prior to asking industry to make the effort to produce implementations.

Box vs. ISA Standardization: The Army is standardizing on the box level as well as the ISA level; the Air Force is only standardizing on the ISA level. The Army plans to purchase standard implementations of Nebula with fixed peripheral interfaces, main memory, and processor power. Thus, the Air Force needs more flexibility in the I/O architecture -- in particular: the types of interfaces that the IOCs will support, the ease of extending the IOC instruction set to support new interfaces, the number of IOCs that can be attached, the support for directly connected devices. The Air Force may have more interest than the Army in other

areas -- for instance: tightly coupled multiprocessing, and putting some provisions for fault-tolerance at the ISA level rather than relegating it to the box level.

Support for other HOL's: The Air Force has a requirement to support applications written in JOVIAL J73 (its current language standard) and to support AI applications requiring an undetermined HOL. Difficulties in efficiently supporting these languages, as brought out in the reports of the reviewers, could increase the Air Force's life-cycle costs of adopting Nebula as a standard -- either through program inefficiencies or by increasing the number of waviers requested to not use MIL-STD-1862A for many applications. Since the Army does not have a similar investment in applications written in standard languages or apparent concern for the demands of AI applications, there is less concern in the MCF program about the use of non-Ada languages (like JOVIAL) on Nebula computers.

The MCF Program Schedule: Since the time that the MCF program started in 1975 with little funding and a very limited staff, it has increased enormously in size and support. The increased emphasis within DoD for standardization of computer systems combined with the Army's lack of any other program for standardizing on a computer architecture has undoubtedly served as much of the driving force. With the failure to acquire a commerical architecture and the development of the Ada standard, the strategy of the program changed to developing a new architecture which

would run Ada efficiently. Although such a strategy would put a resulting computer in a good position in the time period after Ada becomes available, the requirement to have a standard architecture quickly means that the Nebula standard had to be finalized before Ada was finalized.

The design process covered a period from September, 1979, to May, 1980 when the first version of MIL-STD-1862 was available for review by the EIA. This span of time is exceedingly short to produce a complete specification of a new computer architecture. In the rushed schedule, the designers were evidently forced to ignore many aspects of support for programming languages (Ada in particular), and for support of "side issues" like fault tolerance, virtualiziblity, multiprocessing, etc.5

The reviews that were requested were rather rushed and thus not very comprehensive. Typically, the results of the reviews were used for "fine tuning" and, except for the I/O area, were largely ignored if they pointed up areas needing massive changes to the architecture. This report appears to be the most comprehensive review of Nebula, but there are certainly more areas which require investigation and many problems to resolve before Nebula can be considered a mature standard.

The Air Force does not require that Nebula be completed on a rushed schedule -- it has the 1750 standard and no perceived urgent need for a 32-bit ISA. The Air Force can move into a new architecture at a more reasonable pace -- adjusting to the Ada schedule as necessary, considering uses, going through an adequate review process, and ensuring the

---

5 The version of the specification published in July, 1981, finally removed most of the significant errors from the section dealing with the I/O interfaces.

architecture meets its requirements. However, to follow such a plan would apparently require the Air Force to discontinue efforts with Nebula and drop the attractive possibility of a shared standard.

Nebula Achievement of ISA and AFSC-HLSS goals

Instruction Set Architecture Goals: The ISA standardization effort has generated a number of goals. These goals are that MIL-STD-1862A should be:

* Implementable on a family of machines with a wide range of processing power
* An efficient host for implementing MCF HOL's (e. g. Ada)
* An efficient base for implementing MCF communications hardware protocols (e. g. 1553B, RS-232)
* A good target for implementation of a wide range of applications (e. g. real-time systems, data base systems, CCCI applications).
* Reduce the visibility of the hardware to the software.6

An ISA standard that is to be evaluated against these goals must be complete, or its suitability for certain applications will be based on an impression rather than knowledge of how a machine conforming to the ISA actually operates. Points raised earlier in this and many of the other

---

6 Szewerenko, L., Dietz, W., Ward, F., "Nebula, A New Architecture and its Relationship to Computer Hardware," Computer, Vol. 14, No. 2, February, 1981.

reviews indicate that the Nebula designers did not have the time to fully address the extent to which a hardware operation becomes visible when the software must handle exceptional conditions. This issue of completeness of the standard is discussed below from a viewpoint not mentioned by the other reviewers.

AFSC - HLSS Program Goals: The Air Force System Command (AFSC) has a High Level System Standardization (HLSS) Program to expand the standardization of computer resources across Command programs. The goal of this program is to implement the efficient execution of Ada and very high level programming languages in a standard way by 1990. The expected benefits of this program are:

* Reduced life cycle software costs.
* Reduced logistics and training support.
* Both application and support computer programs may be used on multiple projects.
* Use of standard ISA's to facilitate hardware competition.
* Increased portability of computer programs.

The last goal, that of program portability, plays a central role in many of the other goals. In particular, life cycle costs and use of computer programs on multiple projects are tied very closely to program portability. To a certain extent, logistics and training support are also affected by portability. Again, one of the major problems raised by the

review on the portability of programs written for Nebula[7] was the lack of completeness found in the Nebula standard.

Completeness of the Nebula Standard: The requirement for completeness is essential for making the standard work. The notion of completeness being used by the designers differs from the goals stated at the beginning of MIL-STD-1862A. Section 1.2 of the standard states that the purpose of the document is to define the Nebula Architecture "with sufficient precision to permit independent implementations of this architecture that execute identical programs in the identical manner."[8] This might be read to imply that all programs must execute identically on independent implementations. It is clear from the content of the document and discussions with the Nebula designers that this implication is not the intent of the document. The completeness goals of the designers are apparently that: assumptions the programmer is allowed to make about the machine are completely specified. It is not the case that all the behavior the programmer may observe of the machine is completely specified, only that there exist programs which can be written which will execute identically on independent implementations.

The advantages of this approach are twofold. First, the specification may be made simpler and shorter. It has a better chance of being understandable and not loaded with burdensome detail. Secondly, it increases the variety of hardware technologies and performance require-

---

7 Worona, S., Writing Portable Programs for the Nebula ISA, this report.

8 MILITARY STANDARD, Nebula Instruction Set Architecture, MIL-STD-1862A, Date TBD, pg. 1.

ments that implementors may provide. Fewer restrictions are placed on the the implementors and chances of eliminating desirable implementations are reduced.

However, this approach to completeness does have drawbacks. To write programs that are portable across Nebula family members, the burden is on the programmer not to use any information not in the specification. With a complex specification this burden grows greater. Programmers are prone to make assumptions based on readily attainable information about the observable behavior of their machine, assumptions that are not part of the specifications. The programmer writing portable Nebula code must have an intimate knowlege of the MIL-STD-1862A; intimate knowlege of a particular Nebula implementation is actually undesirable. Another drawback of the approach is that debugging and testing code is not an acceptable way to gain confidence in the reliability of that code across different Nebula family members -- this is true whether the family members differ by originating from different suppliers, differ by using different technology, or by resulting from technology insertion on an older family member. Despite the drawbacks of this approach it appears to be the approach taken by the Nebula designers. It was apparently felt that achieving the goals of a wide variety of hardware implementations outweigh the added burden on achieving machine code portability.

The above goals have some strong implications for MIL-STD-1862A as a specification. MIL-STD-1862A forms the most constant interface between the hardware implementors and those writing machine level software. If

the specification is unclear or incomplete in describing to either party the required behavior of the machine, the goals of the standardization programs will be thwarted. In an environment with multiple suppliers independently designing machines conforming to the current version of MIL-STD-1862A, it is unlikely that Nebula software could be easily transported from one machine to the other.9 A specification that is open to inconsistent interpretations forces one to choose between the goals of software transportability, and those of multiple suppliers or meaningful competition between suppliers.

SUMMARY

This paper has been a collection of issues that arose in evaluating the architecture, but did not fall under any of the other reports done or was not included in them for various reasons. Many are, in a sense, minor issues or issues with obvious solutions. However, they are issues that need to be addressed. This was not meant to be a definitive list of such problems. As the other reports indicate the need for several changes to Nebula, this report indicates Nebula's need for further review.

---

9 It should be noted that testing code on all existing members of the Nebula family is not a solution to the problem of machine code transportability across family members. The forseen Air Force environment allows multiple suppliers and competition over the life-cycle of the system. Also, technology insertion will be likely to change the behavior of Nebula implementations -- thus new family members will be always coming along for which confidence in software gained from testing is not transferable.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.

END

FILMED

4-85

DTIC